Date of acceptance Grade

Instructor
Arto Klami

Semi-supervised detection of industrial fouling using ultrasound

Chang Rajani

Helsinki August 16, 2018

M. Sc. Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Chang Rajani

Työn nimi — Arbetets titel — Title
Semi-supervised detection of industrial fouling using ultrasound

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M. Sc. Thesis	August 16, 2018	54 pages + 0 appendix pages

Tiivistelmä — Referat — Abstract

Fouling is a large scale problem in industrial equipment such as heat exchangers or pipes, used in factories, ships, airplanes, etc. Traditionally, such equipment is cleaned using sandblasting, chemicals or mechanical methods, all of which require halting the process, which is costly.

Recently, high-power ultrasound has become a viable option to these methods. In ultrasonic cleaning ultrasound is projected into the equipment from the outside, which means that the equipment does not need to be halted to perform cleaning. While the cleaning itself is not invasive in nature, in most cases vision cannot be used to determine whether cleaning is actually necessary or not. What remains is to have such a method that is also non-invasive.

It is possible to use ultrasound as a kind of a radar to detect whether or not fouling is present, and this has been attempted in previous literature. However, until now, such methods have required extensive manual calculation and knowledge of the physical properties of the setup.

We present the first ever system to concurrently clean and detect industrial fouling using ultrasound and deep learning. Our method does not rely on specific properties of the equipment, allowing it to generalize to large industrial processes where it is not practical to calculate or simulate the cleaning scenario. To this end, we extend existing literature on semi-supervised learning by presenting algorithms used to learn from a monotonic process, and model the high-dimensional signal data using a convolutional neural network that is highly robust to temporal variance. This thesis presents the machine learning solution behind the system, and the cleaning components are provided by Altum Technologies.

Further, we explore methods to detect and counter the so-called *domain shift* that occurs when experimenting in the physical world, and provide experimental evidence that our methods work in practice.

ACM Computing Classification System (CCS):
CCS → Computing methodologies → Machine learning → Learning settings → Semi-supervised learning settings

Avainsanat — Nyckelord — Keywords
machine learning, deep learning, signal processing, ultrasound

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library

Muita tietoja — övriga uppgifter — Additional information

Contents

Acknowledgements

This work was conducted in close collaboration between Altum Technologies and Helsinki Institute for Information Technology HIIT, with Altum providing the laboratory, equipment and knowledge required for working with ultrasound. I would like to thank the whole Altum team for working with me tirelessly, and especially Timo Rauhala and Kasper Peterzéns for both their help with running experiments together, and their enthusiasm.

I would also like to thank my instructor Arto Klami for his patience and guidance during both the experimental and the writing parts of this work. Finally, I thank my friends and family, especially my girlfriend Sonja, for their continued support and advice.

In Helsinki, Finland on the 15th of August, 2018
Chang Rajani

1 Introduction

1.1 Motivation

In many industrial processes it is common for fouling to develop on the inside of pipes and heat exchangers that causes significant outages and costs in the industry. For example, in the dairy industry during ultra-pasteurization of milk, heat exchangers might incur a buildup of milk protein [1]. Traditionally, cleaning performed by halting the process and applying chemical, mechanical or blasting to remove the fouling. These methods have to be performed manually by professionals, which costs money, not to mention the cost of halting the equipment and not producing anything while cleaning is taking place.

Recently, a new non-invasive cleaning method has emerged: high-power ultrasound developed by Altum Technologies [2]. In this method ultrasound is emitted from outside of the equipment to dislodge the fouling that tends to stick to surfaces and accumulate. The use of ultrasound does not require opening the equipment, and thus, the industrial process does not have to be stopped to accommodate it.

Naive application of such technology is to continuously perform cleaning to make sure that the equipment is clean at all times. Ideally, however, we would like to halt cleaning automatically when the equipment is clean to save energy and avoid stressing the cleaning apparatus. However, automatic monitoring for fouling can be difficult – for example in the case of pipes it is generally not possible to have vision inside without costly interruption of the process.

Researchers have previously explored the use of ultrasound in detecting fouling without intervention with good performance in specific tasks where plenty of a-priori information of the equipment is available [1]. In such work features calculated based on the physical properties of the materials used in the construction of the setup are fed into a generic machine learning model to obtain good results on that particular task. However, generalizing into arbitrary structures is not practical using features purely calculated by experts. As the internal structure of pipes, for instance, becomes more complex, the accuracy of the calculations suffers. In addition, they need to be performed for each setup separately.

Ultrasonic signals are measured by using a transducer in pulse-echo mode, emitting a delta spike that is captured after reflecting from the various inner and outer surfaces of the equipment, and passing through, in most cases, liquid. This procedure yields a high-dimensional signal that we are interested using for determining the presence of fouling. Instead of performing further calculation to deduce features from this signal however, we are interested in learning directly from the *raw* ultrasonic signal. That is, given as input a signal $\mathbf{x} \in \mathbb{R}^D$, the output of our model should be whether the equipment is fouled or not.

1.2 Contributions

The goal of this thesis is to create a machine learning solution for the problem of ultrasonic fouling detection. It also goes into depth in terms of the challenges involved when applying machine learning models in real-world signal data, and presents methods for tackling those issues. We are in particular interested in such methods that are viable in practice.

Detecting fouling with raw ultrasound is especially difficult for two reasons. First, the use of very high dimensional audio as input requires models that can cope with such data. The latest trend in machine learning to solve complex tasks like this one is to use so-called *deep learning models* – highly flexible neural networks with multiple *layers* of nodes [3]. Their success is in part because they are so efficient to train using modern parallel computing tools, and because we can explicitly encode the nature of the input, in this case the fact that the input is an audio signal, into the model itself. This makes deep learning models an attractive choice for learning from high-dimensional raw signal.

The second problem is that it is difficult to provide ground-truth labels for each captured signal – that is, since looking inside the equipment is generally out of the question, we cannot say whether or not the equipment is clean. In this work we present new techniques for fouling detection based on only the raw signal from a high-frequency ultrasound transducer. We show that we can detect fouling with high confidence even when specific properties of the fouling and type of pipe, internal structure, etc, are not known. To address the problem of not having hand-labeled training data, the presented algorithms require only a small amount of supervision for learning.

Instead of having human annotations, we assume knowledge of labels for a small subset of the data, which is easily satisfied in this scenario, while treating the rest as unlabeled. In particular, we build upon the work of Lee [4] and Longi et al. [5]. The former introduced a simple method for semi-supervised learning called *pseudo-labels*, and the latter extended it by assigning additional constraints in the form of temporal continuity in wifi interference detection. We extend these techniques by introducing the concept of monotonicity into the framework of pseudo-labels, detailed in section 2.3.

Using these methods and laboratory equipment where cleaning is performed simultaneously with detection, we show that ultrasonic fouling detection can be made practical without explicit labeling or calculations that utilize a priori information. To this end, two different scenarios are presented and evaluated. In one, the detection is performed after cleaning is complete, based on a single run from fouled to clean. In the other, previous runs are used to learn a real-time detector. A scientific publication building on this part of the work has been published in IEEE Workshop on Machine Learning for Signal processing [6], and this thesis extends the work published by looking at the bigger picture behind the solution.

We also examine tools and methods for visualizing the datasets that are explored in

this thesis. For example, using non-linear dimensionality reduction algorithms, we come to the conclusion that there exists a domain shift between different experiments performed at different times.

Finally, we review and implement methods for generalizing a model trained using one laboratory setup to another setup, a technique called *domain adaptation*. We explore different domain adaptation algorithms and evaluate their performance on a synthetic task where the simulated physical properties of the setup change between the training and prediction scenarios.

1.3 Structure

This thesis is structured as follows. Chapter 2 introduces basic machine learning concepts, deep learning models and the methods used to train them, as well as algorithms used for domain adaptation. Chapter 3 introduces the problem of ultrasonic industrial fouling detection, introduces our laboratory test setup, and visualizes the data collected. In Chapter 4 we present our algorithms and models for detecting fouling in a semi-supervised fashion. Finally, Chapter 5 delves into the empirical experiments performed to validate the efficacy of our models in the real world.

2 Background

2.1 Learning tasks

The world is full of problems that are solvable using computers. Some, such as sorting an array, can be solved exactly using algorithms for which it can be proven that the result will always be correct. These algorithms work based on rules written by the programmer for a specific task. On the other hand, some problems (to the best of our knowledge) are not solvable exactly, possibly because it is extremely difficult to write rules for them. For instance, the task of object recognition entails, when given an image as input, to give as output the names of all the different objects, such as cars, birds or bow ties, in that image. In such a setting, the result will most certainly not be always correct, as not even a human can complete this task with one hundred percent accuracy, but we might still be able to create a program that does it well – in fact, even better than humans [7].

Such programs often employ *machine learning* algorithms, that instead of using explicit, human-written rules to make decisions, *learn* their operation from data. In the case of object detection, given a set of data that contains images and human-made annotations of what is in the image, the algorithm has to create a *model* of the phenomenon, so that given a new image it can produce said results.

More generally, in machine learning we are interested in using observations of a phenomenon to construct a model that not only *fits* the already-observed data, but *generalizes* to new, unseen instances of data. This is an important distinction, because without it machine learning would be a simple task of optimization – the act of fitting our chosen model to the given observations as well as possible.

Machine learning is generally split into two main groups, unsupervised and supervised learning. In *unsupervised learning* our task is to learn some sort of a structure from the given data, without guidance from a human or other entity. The problem is difficult, because there might be plenty of different structures in any particular set of data, and the resulting model might not entail anything useful. Examples include *clustering*, where we wish to assign each data point to a cluster that contains points that are similar by some metric, and *dimensionality reduction*, where the aim is to reduce the dimensionality of the data while preserving important properties.

In *supervised learning*, on the other hand, our task is somewhat easier. The given data consists of not only the observations, but annotations that specify what the output should be for a particular sample. The task is then to learn a model that predicts such outputs correctly for new data. More specifically, supervised learning means minimizing a loss function $\mathcal{L}()$ with respect to some model $M_\theta()$ that depends on unknown parameters θ. Given a dataset $\mathcal{D}$ of labeled data points $\mathbf{x}, y$ our task is then to find θ such that

$$\arg\min_{\theta} \mathbb{E}_{p(\mathbf{x},y)} \; \mathcal{L}(M_\theta(\mathbf{x}), y) \tag{1}$$

The expectation is defined over the whole distribution of $\mathbf{x}, y$ pairs, which characterizes the fact that we are interested in not just fitting our training data, but other data points from the same distribution as well. In practice we are usually happy with finding a local minima, but sometimes a global optimum is also achievable.

Many approaches lie in between these two worlds, and can be generally called *semi-supervised machine learning*. In such a setting a part of the data might be annotated, but usually most of it is not. We discuss this scenario in section 2.3.

Tasks such as classification and regression where the output is discrete or continuous, respectively, are supervised learning problems. The example of detecting objects from images is a task of classification, because the different objects the model can classify the image into are discrete. On the other hand, continuous targets are found in problems such as stock market price prediction.

In probabilistic terms, we can also think of supervised machine learning as modelling the unknown true joint distribution of the data and labels $P(X, Y)$. In this interpretation *generative* models seek to find a function that best approximates the entire data-giving distribution, while *discriminative* models model $P(Y|X)$, which in the case of classification means finding the decision boundary that separates the target classes. This interpretation of machine learning will be useful in Section 2.6.

In this chapter, along with looking at basic machine learning algorithms, tasks, techniques and models, we go deep into a specific subfield of machine learning called *domain adaptation*, presented in Section 2.6. The emphasis placed on this technique is a result of our ultimate goal – the real world experiments presented in this thesis entail a strong need for it, as we will see in Section 5.

For more information about the concepts presented here, please refer to, for example, [8].

2.2 Generalization

As mentioned, since machine learning is about generalizing to unseen data, fitting the training data too well while generalizing poorly is called *over-fitting*, and is in most cases not desirable. Apart from improving generalization, it is also important to be able to evaluate it as accurately as possible. In this section we look into techniques for both, starting with evaluation, and moving on into improvement.

2.2.1 Evaluating generalization

Perhaps the most widely used technique to evaluate how well a model generalizes is use of a separate test set. The given observations are split into two sets, one for training the model, called the *training set*, and one for evaluation, called the *test set*. Once a model has been trained on the training set, it can be evaluated on the test set, which contains data it has not seen in training. The test set error then

serves as an approximation for the true generalization error, that is,

$$\frac{1}{N}\sum_{\mathbf{x}\in\mathcal{D}_{\text{test}}}\mathcal{L}(\mathbf{x},y)\approx\mathbb{E}_{p(\mathbf{x},y)}\mathcal{L}(\mathbf{x},y)$$

In other words, the test set loss approximates the expected loss for all possible inputs that might be fed into the model.

Another popular technique for evaluating how well the model generalizes is *cross-validation*. In the traditional two-way split just discussed, much of the data cannot be used for training at all. Cross-validation addresses this flaw by splitting the data into different parts, training and validating with each split, and then averaging the results to get an estimate for generalization.

Different kinds of cross-validation exist, with some of the most common ones being k-fold cross-validation and leave-one-out cross-validation. In the former the data is randomly split into k parts, each part is consecutively used as the test set, while the other parts are used for training. The results are then usually averaged to yield a score. Leave-one-out works similarly, except each sample is validated on separately, while training on the other samples.

When working with multiple separate real-world experiments, where a single experiment produces a set of samples, it can make sense to cross-validate by leaving each *experiment* out in turn. This measures generalization well if the model needs to generalize to new experiments based on previous ones.

2.2.2 Improving generalization

Equally important to detecting poor generalization is preventing it. For this purpose, a widely used method exists called *regularization* [8]. Regularization applies a penalty to the scale of the parameters in our model, which ensures that the model is a simpler function of its input. Simpler models should be preferred over complex ones because small changes in the data would change the parameters of the model a lot [8].

One observation is that if the parameters are allowed to be really large, a single feature can be used to make the entire decision [9]. If that feature is not available during inference, then the model cannot generalize. Therefore, preferring parameters to be closer to zero should result in a model that uses all features in making the decision. To address this, given a loss function $\mathcal{L}(M_\theta(x),y)$ that depends on the parameters θ of the model $M()$, we can use l_2-regularization, given by

$$\mathcal{L}_{l_2}=\mathcal{L}(M_\theta(\mathbf{x}),y)+\lambda\sum_{w\in\theta}w^2.$$

This penalty over the weights prefers solutions where most parameters are evenly near zero. The hyperparameter λ is used to control the strength of the regularization, and is often found by cross-validation [8].

Another popular regularization term is the l_1 regularizer, which instead thresholds parameters that are nearly zero to zero. l_1 results from penalizing the absolute sum of the parameters instead of the squared sum, and is given by

$$\mathcal{L}_{l_1} = \mathcal{L}(M_\theta(\mathbf{x}), y) + \lambda \sum_{w \in \theta} |w|.$$

This regularization induces sparsity, i.e perfers solutions where some parameters are zero.

The amount of data available for training also influences generalization, as more data generally means less overfitting [8]. However, it is often difficult to generate more data, especially in most supervised learning tasks, where it requires labeling additional samples, usually by hand.

2.3 Semi-supervised learning

In between the two main approaches to machine learning, supervised and unsupervised learning, lies a natural continuation of both. In semi-supervised learning, both labeled and unlabeled available data is used in making predictions. The question becomes how to make use of the latter, since usually unlabeled data outnumbers the available labeled data [10]. Importantly, the aim is to learn a better model than what is possible by simply using the labeled data, or by using all the data in an unsupervised manner, ignoring the labels. This is especially important since there is often a significant cost associated with generating ground-truth labels for a specific task. As a practical example, Rasmus et al. [11] used an auxillary objective of denoising internal model representations of the data to improve classification accuracy. Their method achived over 99% accuracy on the MNIST dataset with just 100 labeled (the entire dataset is 50000 training samples), using the rest of the dataset without labels.

Denote by $\mathcal{D}_l$ the labeled training samples, and by $\mathcal{D}_u$ the ones without a label. A natural way of utilising the data with no labels is then to approximate labels $\hat{y}$ for the samples $\mathbf{x} \in \mathcal{D}_u$ and then train using the combined dataset $\mathcal{D}_u \cup \mathcal{D}_l$. One way to approximate $\hat{y}$ is by assuming that samples that are near each other probably share the same label. Then, for each unlabeled sample $\mathbf{x}_i$ we can find the nearest sample by some distance measure

$$\hat{y}_i = C[\arg\min_j \text{dist}(\boldsymbol{x_i}, \boldsymbol{x_j})],$$

where the notation $C[j]$ denotes the class of the jth sample. For example, when measuring the distance as euclidean distance we would set

$$\text{dist}_{ij} = \sqrt{\sum_d (x_{id} - x_{jd})^2}.$$

This approach is more ambiguous if we are working with a regression problem instead of classification, since in many cases it is not likely that two samples share the exact same label.

Another technique for semi-supervised learning is known as *pseudo-labeling*, proposed by Lee [4]. In this approach a model is trained on the labeled training data, and used to predict labels for the unlabeled samples. We then use all the samples to train a new model. The process is repeated for a fixed number of iterations or until convergence. The idea is that if the model is unsure about the label of a specific sample, the pseudo-label assignment reduces that uncertainty, possibly steering it in the correct direction. Lee used the model for semi-supervised classification of hand-written digits, and achieved almost 90 percent accuracy with just 100 samples.

Algorithm 1 gives the pseudo-code for such an approach. This version of the algorithm keeps the labels we know for sure fixed, but if we know that they are also noisy, we could let it change those as well by predicting new labels for the entire dataset at each iteration.

Algorithm 1 Pseudolabeling

1: $\mathcal{D}_l \leftarrow$ initial labeled samples
2: Optimize initial parameters θ of model using $\mathcal{D}_l$
3: **while** not converged **do**
4: Predict labels for $\mathcal{D}_u$
5: $\theta_{t+1} \leftarrow [\theta_t$ optimized using $\mathcal{D}_l \cup \mathcal{D}_u]$

Lee also proposes a way of taking into account the confidence of each unlabeled sample in the learning phase. Denote by N_l and N_u the amount of labeled and unlabeled training samples, respectively, by y and $\hat{y}$ the true labels and pseudolabels, and by z the prediction given by the model for a given sample. The proposed method then corresponds to the following optimization objective

$$\sum^{N_l} \mathcal{L}(z, y) + \alpha(t) \sum^{N_u} \mathcal{L}(z, \hat{y})$$

where $\alpha(t)$ is a deterministically increased weight for the t:th iteration. $\alpha(t)$ controls how much we trust the pseudolabels – low α means getting the true labeled examples right is much more important. Lee says that too high α might disturb the training of the labeled data, while a too low one causes the unlabeled data to be of no use [4].

2.4 Deep learning

In part inspired by how the brain works, deep learning models learn a stacked, hierarchical representation of the data, while solving the task at hand at the same time [3]. By learning to represent the data for the given task, one can avoid hand-tuning features. Deep learning models are mostly implemented as feedforward networks

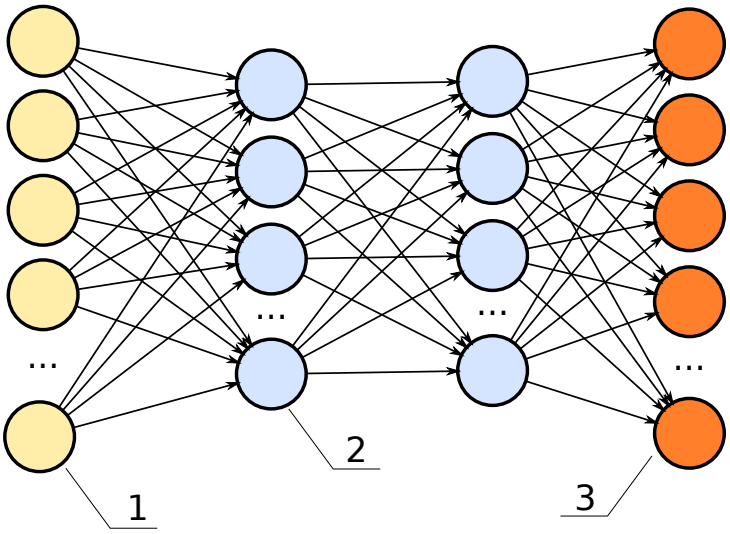

Figure 1: An illustration of a deep neural network. (1) the inputs of the network, (2) the hidden layers and (3) the outputs. Best viewed in colour. Illustration by Zufzzi, released for public domain

that are efficient to train using modern hardware, and the gradient descent algorithm.

For a number of years now deep learning has been dominating the field of machine learning. After significant advances in parallel processing power, learning algorithms, and increased availability of data, many deep learning techniques have been rejuvenated from the 90s, and much research on focuses on it today. Deep learning models are very flexible, and therefore easy to apply to a variety of tasks, perhaps most famously in the ImageNet image recognition competition in 2012 which launched deep learning, or more specifically, convolutional neural networks, into the mainstream [3, 12].

Traditionally, to use large inputs with structure, such as audio or video, in machine learning models, one has to devise elaborate ways of extracting features from the data that are then fed into the model. For example, before the 2010s many algorithms for image classification relied on features extracted in advance – that is, before a learning algorithm is run – from an image [3], using an algorithm such as Scale-invariant Feature Transform (SIFT) or variants [13], which meant that the features extracted were most of the time task-independent. On the other hand, human domain experts can be drafted to create task-specific features, but this requires a significant amount of work and time.

It is also natural to think that different tasks require different types of features to be extracted from an image. For example, for detecting square-shaped objects it might be very useful to extract edges from images, but the same would probably

not apply to circle-shaped objects. Deep learning addresses these points by learning features directly from data, instead of using generic ones [3].

2.4.1 Anatomy

Deep learning models (also called artificial neural networks, or ANNs) consist of stacked layers of nodes, also called neurons. Each neuron computes a multiplication with its input, and produces an output which is then sent to the next layer. Layers also generally have an activation function that is used to make the output non-linear, as well as a bias term. The data comes into the network, moves through each layer, computing multiplications and additions layer by layer, and out comes the prediction. This modularity is key to the ease at which such networks can be implemented – layers can be stacked on top of each other like lego blocks, and software packages provide efficient implementations for most layers. See Figure 1 for an illustration of an ANN.

Compared to traditional supervised methods, deep learning models usually take the entire raw sample, such as image or signal, as input, while learning features specifically useful for the task at hand. The multiple stacked layers of an ANN each learn features of decreasing detail and increasing abstraction – that is, the further in the network a layer is, the more complex are its learned features [14].

There exist many different types of layers, but the most basic one is the *dense layer* that consists of a set of parameters $\boldsymbol{W}$, biases $\boldsymbol{b}$ and an activation function f. It takes its input $\mathbf{x} \in \mathbb{R}^d$ and computes the result

$$\boldsymbol{y} = f(\boldsymbol{W}\mathbf{x} + \boldsymbol{b})$$

which is then used as input for the next layer [3]. The amount of columns in the weight matrix $\boldsymbol{W}$ is a hyper parameter called the amount of nodes. More nodes increase the *width* of the network, while more layers increase its *depth*. The word deep learning comes from the fact that in general, the more layers in a network, the more learnable parameters it has, in which case we call the network deeper.

The activation function ensures that the layer constitutes a non-linear transformation of its inputs. Without it, any network is simply equivalent to one with a single hidden layer with different weights. Any differentiable monotonic function is suitable for use as an activation, but common choices for activation functions include the ReLU $f(x) = max(x, 0)$ and the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ [3].

2.4.2 Learning

Neural networks are typically trained with the backpropagation algorithm and stochastic gradient descent [14]. The idea is that all gradients of the weights with respect

to the loss can be calculated efficiently using the chain rule of derivation, and then it suffices to update the weights in the opposite direction of the corresponding gradient.

Let us assume we wish to minimize the loss $\mathcal{L}(\theta)$ that a model makes with parameters θ. We start with initial random guess θ_0. We then repeat the following steps until convergence

1. Evaluate gradient $\nabla_\theta \mathcal{L}(\theta)$ at current θ_t
2. Adjust parameters $\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta \mathcal{L}(\theta)$

Here α is called the *learning rate* or *step size*, that controls the size of the update. Larger steps require less function evaluations, but might overstep the minima and arrive at a poor solution. There are also ways of adjusting the learning rate dynamically after each iteration. For example, it is common to decrease the learning rate as the learning progresses to avoid jumps that are too large [14]. This can be done either by decreasing it altogether or for each individual parameter, depending on some calculation based on the previous gradients.

An example of the latter approach is Adagrad [15], where we adjust the gradient update for each parameter θ_i at each timestep t with

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}} \cdot \nabla_\theta \mathcal{L}(\theta_{t,i})$$

where ϵ is a small constant to prevent division by zero, $G \in \mathbb{R}^{d \times d}$ is a diagonal matrix, and

$$G_{t,ii} = \sum_{j=1}^{t} (\nabla_{\theta_i} \mathcal{L}(\theta_{j,i}))^2$$

In other words, Adagrad adjusts the learning rate for each parameter based on the cumulative sum of previous gradients up to that point. The idea is that if the data is sparse, we want to assign large learning rates to rare features that are informative, while keeping the learning rate low for frequently appearing features. Unfortunately, since the denominator keeps increasing, the learning rate will keep decreasing towards zero at every step, which may cause problems since at some point the algorithm will not be able to learn anything new.

To address this flaw, further methods have been proposed. A popular alternative to Adagrad, called Adam [16], works by only keeping previous gradients of a window of certain length, so that the denominator cannot grow indefinitely. More specifically, it computes an exponentially decaying average of previous gradients, as well as previous squared gradients (the variance), and uses them to adjust the learning rate for each parameter. Denote by $g_t = \nabla_\theta \mathcal{L}(\theta_t)$ the gradient of the loss at timestep t. Now, the exponential decaying average is defined as

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

where $m_0 = 0$ and $\beta_1 \in [0, 1)$ is the exponential decay rate for the running mean, which the authors recommend setting to 0.9. Further, the variance is approximated by

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

where β_2 is again an exponential decay rate hyperparameter, which the authors set to 0.999. As the initial values for m_0 and v_0 as initialized as zero-vectors, using these estimates for the mean and variance of previous gradients would bias them towards zero. To correct this bias, the authors propose the following additions to the above equations,

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally, the update rule is described as

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

In general, we can apply gradient descent on either the entire dataset at a time (normal gradient descent) or a randomly chosen smaller *mini-batch* at a time (stochastic gradient descent). A bigger batch tends to approximate the underlying phenomenon better, but is also slower to compute and the necessary weights and gradients might not fit in memory. On the other hand, a smaller batch is a worse approximation but faster to compute. It has also been shown that models trained with smaller batch sizes tend to generalize better [17], although the reasons for this are not entirely clear.

2.4.3 Backpropagation

To efficiently evaluate the gradient for each weight, an algorithm called backpropagation (BP) is most often used [14]. Evaluating the network on a batch of samples is called the *forward pass*, where as computing the gradients using BP is called the *backward pass*. One iteration of training then consists of a single forward pass, a single backward pass, and the weight update.

The objective of the BP algorithm is to find the gradient of each weight in the network being trained with respect to the loss function $\mathcal{L}$ [14], after which one usually takes a small step towards the inverse of the gradient to reduce the loss (called gradient descent) [3]. To compute each gradient efficiently, one notes that if we have already computed the gradients for the following layer, we can apply the chain rule of derivation to derive the gradients for that layer. This is why the algorithm proceeds backwards, starting from the end of the network.

More concretely, let us look at the example network in Figure 2. To calculate the gradient of the weight w_8 w.r.t the loss, we walk back from the output to the desired

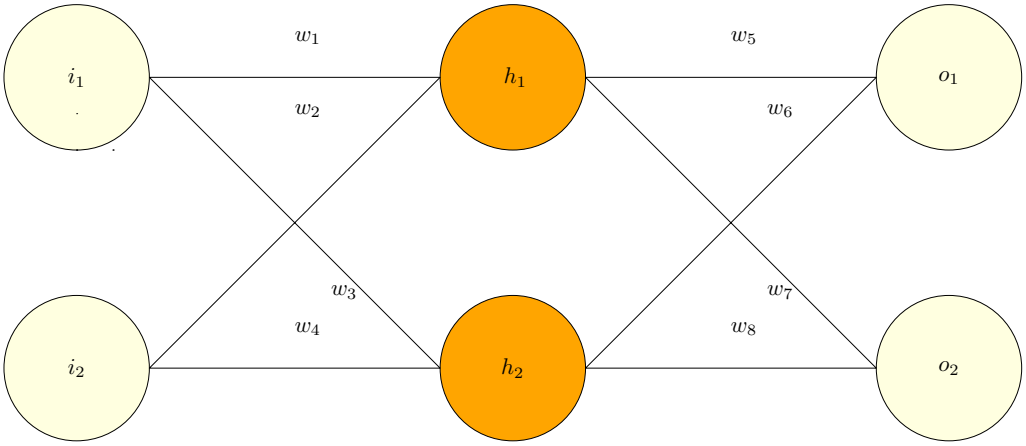

Figure 2: A simple neural network with one dense hidden layer

weight, computing gradients in between. Here we assume the sigmoid function as the activation for all nodes, but the same principle applies in general to all models that can be differentiated. We can apply the chain rule to yield

$$\frac{\partial \mathcal{L}}{\partial w_8} = \frac{\partial \mathcal{L}}{\partial \sigma(o_2)} \cdot \frac{\partial \sigma(o_2)}{\partial o_2} \cdot \frac{\partial o_2}{\partial w_8}.$$

Each piece of this equation can be computed assuming we are at an output node (as is the case for w_8), or if we have already computed the gradients for the next layer. The quantity $\frac{\partial \mathcal{L}}{\partial \sigma(o_2)}$ is simply the derivative of the loss with respect to the output of the network. On the other hand,

$$\frac{\partial \sigma(o_2)}{\partial o_2} = \sigma(o_2)(1 - \sigma(o_2))$$

is the derivative of the sigmoid function. Finally,

$$\frac{\partial o_2}{\partial w_8} = \frac{\partial w_8 o_2}{\partial w_8} = o_2.$$

The other gradients can be computed similarly.

2.4.4 Loss functions

The loss function $\mathcal{L}$ can be any differentiable function of the input and output, and depends on the task at hand. For regression problems, where the output is any real number, the mean squared error (MSE) is often used, given by

$$\mathcal{L}_{\text{MSE}}(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{2}$$

For binary classification, given two classes $C_1 = 0$ and $C_2 = 1$, we assume that the class label follows a Bernoulli distribution

$$p(y) = \prod_{i=1}^{n} P(y_i = 1)^{y_i} P(y = 0)^{1-y_i}$$

where the probabilities $P(y_i = 1)$ are given by the neural network by constraining its outputs to the [0..1] range for example by taking the sigmoid function at the last layer. Taking the log and noting that $P(y_i = 0) = 1 - P(y_i = 1)$, we get the log-likelihood

$$\begin{aligned} p(y) &= \sum_{i=1}^{n} \log P(y_i = 1)^{y_i} + \log(1 - P(y = 1))^{1-y_i} \\ &= \sum_{i=1}^{n} y_i \log P(y_i = 1) + (1 - y_i) \log(1 - P(y = 1)) \end{aligned}$$

The log-likelihood is a function we would like to *maximize*, but in machine learning we generally deal with losses, so we simply use the negative log-likelihood as the loss function. Replacing $P(y_i = 1)$ with $\hat{y}$, we get what is called the *binary cross-entropy loss* or simply *log-loss*,

$$\mathcal{L}_{\text{BCE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \tag{3}$$

To extend the log-loss to any amount of classes C, one usually changes the neural network so that the output is a vector $\hat{y} \in \mathbb{R}^C$ where each element corresponds to the probability $P(y_i = c), c \in [1..C]$. To treat the output as a probability distribution over classes, that is, to be able to assume $y_i \sim \text{Categorical}(C)$, the outputs need to be normalized so that they sum to one. To do this, it is common to use the *softmax* function as the activation for the last layer, given by

$$\text{softmax}(\boldsymbol{z}) = \frac{e^z}{\sum_c e^c}$$

This ensures that each element of the output vector is a probability. The *categorical cross-entropy loss* is then given by

$$\mathcal{L}_{CE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{n} \sum_{c}^{C} \log P(y_i = c)^{\mathbb{1}[c=y_i]} \tag{4}$$

$$= -\sum_{i=1}^{n} \sum_{c}^{C} \mathbb{1}[c = y_i] \log P(y_i = c) \tag{5}$$

which only penalizes for the prediction of the correct class, and ignores the prediction for other classes.

```
import torch
model = torch.nn.Sequential(
    nn.Linear(128, 256),
    nn.ReLU(),
    nn.Linear(256, 64),
    nn.ReLU(),
    nn.Linear(64, 1),
    nn.Sigmoid()
)
```

Figure 3: A three-layer dense neural network defined in PyTorch. The model has 128 input nodes, two ReLU-activated hidden layers of 256 and 64 nodes, respectively, and a single output node with sigmoid activation. The network is immediately ready for computation, as weight and bias matrices are allocated and initialized, and the forward and backward propagations do not need to be separately written.

2.4.5 Frameworks

In the modern deep learning scene many frameworks for popular programming languages exist to make the task of designing and implementing neural networks easy. These frameworks automatically perform training and inference after the user describes the model and the specific optimization algorithm to be used. They can also automatically distribute the work to multiple CPU cores, processors or GPUs.

Most deep learning frameworks are built on fast matrix algebra libraries that utilize hardware-level SIMD (Single instruction multiple data) instructions that allow extremely efficient vector and matrix operations. Even though typically one interfaces with these libraries from a high-level language such as Python, the underlying libraries are written in fast native code such as FORTRAN or C++. In addition, NVIDIA's CUDA is used to run matrix operations in parallel on GPUs.

A technique called *automatic differentiation* can be used to compute the backward pass automatically [18]. It works by keeping track of changes made to the parameters during the forward pass, and using the chain rule of derivation to compute the gradient based on pre-derived derivatives for common functions. In fact, it is uncommon to explicity use the backpropagation algorithm as defined by the chain rule – instead, most frameworks let one write the forward pass and the framework takes care of gradient calculation.

The most common frameworks in use today are TensorFlow [19] by Google and PyTorch by Facebook [20]. TensorFlow uses static computational graphs which allow it to run extremely fast, but as a downside the whole network needs to be specified before running any training on it. By contrast PyTorch makes use of dynamic graphs which let users temper with the architecture of, for example, a neural network *during runtime*. This helps with prototyping and opens up new ways of implementation. See Figure 3 for a code example of defining a neural network.

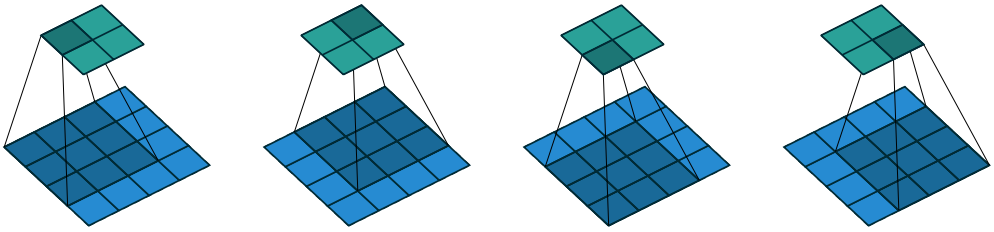

Figure 4: Illustration of the convolution operation in 2D. Green cells correspond to outputs, and blue cells to inputs. The dark green cells are the sums of element-wise products computed from the dark blue cells. Here $k = 3$, $s = 1$ and no padding is used. Adapted from Dumoulin et al. [21].

Frameworks such as these make implementing deep learning models extremely easy, and play a key role in the popularity of deep learning. While previously one had to calculate derivatives by hand, an error-prone and time consuming procedure, now one simply defines the forward pass of the model, and the gradients are calculated automatically, even for the most complex of formulas. In addition, gradient descent optimization algorithms such as Adam [16] or Adagrad [15] are widely available, with the programmer only having to name the algorithm to use it.

2.4.6 Convolutional neural networks

Neural networks made of only dense layers have an important shortcoming when it comes to images, audio and other structured inputs: they do not capture the structure of the data [12, 21]. To feed an image into a dense layer, it has to be flattened from $\mathbf{x} \in \mathbb{R}^{W \times H \times C}$ tensor form into a single vector $\mathbf{x} \in \mathbb{R}^{WHC}$. This means that the model has no way of knowing which pixels are close to each other. While it is theoretically possible to learn to approximate any function using a network of dense layers [22], in practice encoding prior knowledge about the data to the model, i.e using models that explicitly take as input images lets us learn such a function much more efficiently.

Convolutional neural networks (CNNs) introduce two new types of layers into the neural network architecture. The main building block of CNNs is the convolutional layer, which can be thought of as sliding a window, called *filter* or *kernel*, over the input matrix from left to right, top to bottom and computing the element-wise product between the filter weights and the input [21]. This operation is called convolution in signal and image processing literature. The other main layer type is the pooling layer, which reduces the amount of parameters in the model while merging similar features into one [3].

Apart from taking into consideration the shape of the input, a technique called parameter sharing is often used to reduce the amount of parameters in the model. Instead of learning separate filter weights for each position in which the filter is evaluated (convolved), the same filter weights are used for each position [3]. Intuitively,

if a feature is present at some part of the input, it is worthwhile to check if the same feature is present elsewhere as well.

For vector-shaped inputs convolution can be done with vector-shaped filters. Since the input and filters have a single dimension, this is called 1-D convolution – also sometimes called temporal convolution, because when used on audio signal the filter travels over the temporal domain. Arguably the most common form of convolution is 2-D, mostly used on images and other matrix-sized inputs – here both the input and the filters are matrices. Convolution can be defined in the general case for N dimensional inputs [21], but for the remainder of the thesis we will focus on temporal convolution, since we are dealing with audio data. See Figure 4 for an illustration of convolution in 2D.

To properly define convolution we need to define

1. The size of the filters k,
2. How many steps to take when moving the filter, called the *stride* s,
3. Zero-padding on the input, p.

In the one dimensional case the filter size can be thought of as the window that is being slid over the audio signal, while the stride controls how many elements to skip at each point, that is, for $s = 1$, the window moves one element at a time. The larger the stride, the smaller the output, so the stride can be used as a way of reducing the amount of parameters in the model. The stride and filter size are also convenient for encoding a-priori assumptions into the model.

In addition, zero-padding can be applied to each edge of the input. When the input is not divisible by k, the final activations of the filter will "go over" the input, so padding can be added to stop this from happening. One possible use case for padding is to make the output the same size as the input, which will not happen if no padding is applied unless $k_i = 1$. The output of the convolution operation will be the same as the input for even i and odd k when $p = \lfloor k/2 \rfloor$, and is sometimes called "same" or "half" padding [21].

The other main layer present in CNNs is the pooling layer. Pooling takes in a vector or matrix of inputs like convolution, and radically reduces its size by picking a constant to represent a part of it. Common examples of pooling include *max pooling*, which picks the maximum value in a region and *average pooling*, which instead keeps the mean of the range. In effect, this causes the exact location of the computed features to be less significant.

The i:th element of the resulting vector $\boldsymbol{o}$ from a max pooling operation in the 1-D case with stride $s = 1$ and window size k on input $\mathbf{x}$ is given by

$$\boldsymbol{o}_i = \max(\mathbf{x}_{i-\lfloor k/2 \rfloor}, \ldots, \mathbf{x}_{i+\lfloor k/2 \rfloor}).$$

Note that the pooling layer loses information about its input, namely, which element gave the maximum and what the other elements considered were. This is of course

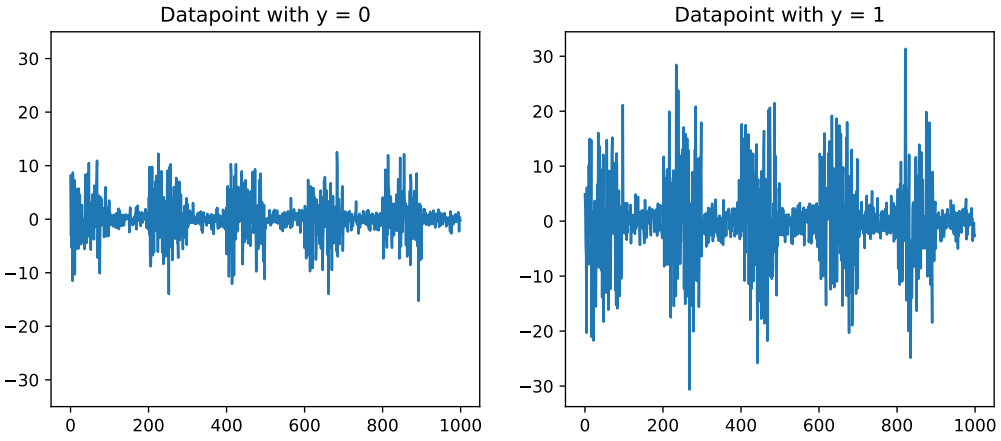

Figure 5: Two examples of the toy dataset $\mathcal{X}$, with different labels, and $D = 1000$ dimensional datapoints. Samples with $y = 1$ use a Gaussian with a larger mean.

intended, as the point is to drastically cut down on the amount of parameters in the model.

2.4.7 An example

To understand the features computed by CNNs, let us examine a toy dataset $\mathcal{X} \in \mathcal{R}^{N \times D}$ created by concatenating independent Gaussian vectors such that every other vector is drawn from $\mathcal{N}(\mathbf{0}, \mathbf{1})$, and the remaining ones are drawn from $\mathcal{N}((c_1, \ldots, c_1), \mathbf{1})$ with probability $p = 0.5$, and from $\mathcal{N}((c_2, \ldots, c_2), \mathbf{1})$ otherwise. We further create a binary classification task by setting the label $y = 0$ for samples where the non-noise part is drawn from the c_1-centered distribution, and $y = 1$ otherwise. In other words, $\mathcal{X}$ consists of intermittent noise and Gaussian data with different means, depending on the label. Figure 5 gives examples of the data, with $c_1 = 5$ and $c_2 = 10$.

We model this dataset with a simple CNN that consists of

1. A ReLU-activated 1D convolutional layer as the input layer, with a single filter of size $k = 10$ and stride $s = 1$.

2. A max-pooling layer with $k = 10$ sized window, and

3. A dense output layer that takes in the convolved and pooled output of the previous layers, and produces a softmax-activated 2-dimensional vector, the probabilities of each class.

Note that replacing the convolutional layer with a linear one would result in a model with much more parameters, depending on the choice of nodes for the layer. Even with 10 nodes, the layer would have $1000 \times 10 + 10 = 10010$ (accounting for the bias) free parameters, where as the convolutional layer only has $1 \times k + 1 = 11$ free

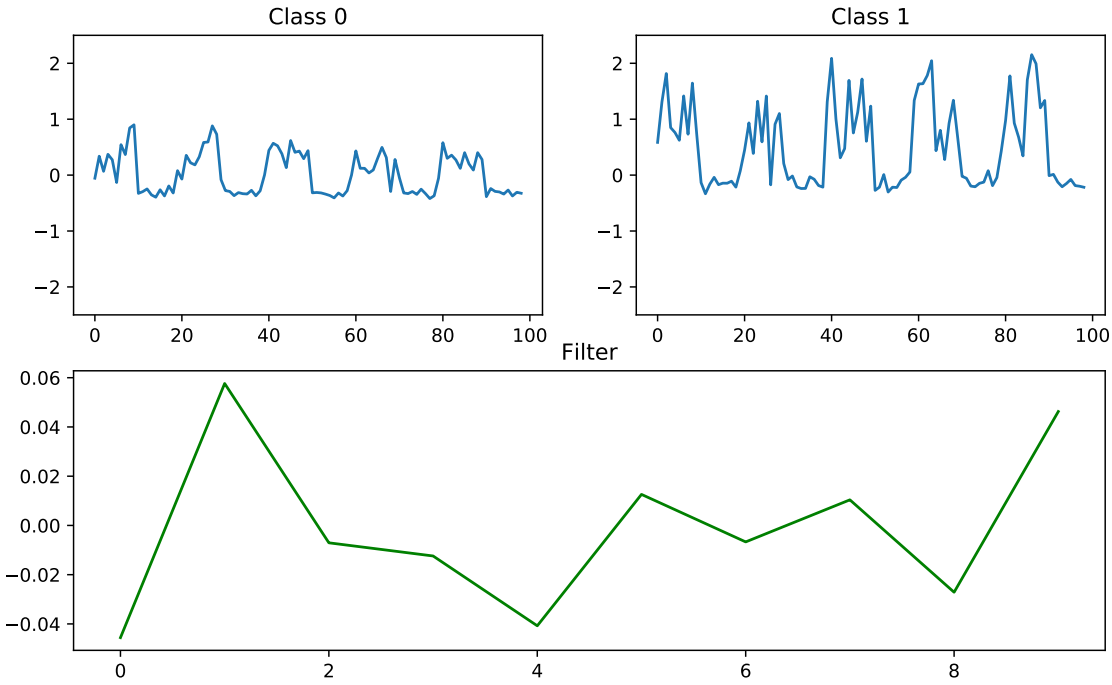

Figure 6: Top: the feature maps computed by the trained network after the max pooling layer, on two separate test instances with different classes. Note that the scale of the activation is higher in the case of class 1. Bottom: the filter weights learned in the first layer of the network. We see that the filter emphasizes the first two features, indicating that it detects edges in the input.

parameters. This, coupled with pooling allows for a much simpler model in terms of number of parameters, but a more expressive one in terms of modeling audio data: after all, most temporally nearby features are highly correlated, and we only need the non-noise parts to make the decision.

We train the model by minimizing the cross-entropy loss (Eq 5 using the Adam optimizer presented in Section 2.4.4. Figure 6 shows the feature maps or activations of the convolutional layer followed by the pooling layer, computed from a network that has reached 99% test accuracy. The features computed indicate that the network has learned, correctly, that the difference between the two classes is in the scale of the inputs at certain positions. This also reinforces the intuition that the first two layers act as feature extractors – after their activations the two classes are clearly linearly separable.

In this task almost all hyperparameter settings for the filter sizes, etc, we chosen by hand and worked well. However, if intuition about the task is not strong enough to select hyperparameters, automated strategies for finding good settings exist. A common method is to run a grid search, trying out different alternatives on a linear plane, although recent literature has shown that a random search that tries different hyperparameter settings randomly is usually better [23].

2.4.8 Regularization in deep learning

Since deep neural networks are highly flexible models with huge amounts of parameters, they are prone to overfitting. Generalization can be helped by using huge amounts of training data, but also other, architechtural improvements have been suggested in literature. The most popular ones include Dropout [24] and Batch Normalization (BN) [25].

Dropout is an extremely simple method to implement. It works by during training randomly removing nodes and their incoming and outgoing connections. More specifically, during training, a node that consists of weight w and bias b is left out of the computation of the layer it belongs to with probability p. At test time nothing is removed.

A significant problem in neural networks is that is a single node starts making large mistakes, a local optimum in terms of the loss may be found by having another node correct the mistake. This leads to the two nodes becoming *co-adapted*, which means that neither contribute anything useful towards the actual predictions. One of the reasons for its regularizing effect is that dropout reduces co-adaptation by making it difficult for a node to "rely on" other nodes, since they might not be available for correcting the mistake [24].

Batch Normalization on the other hand both acts as a regularizer and speeds up training. The authors of BN suggested that a key problem in training deep neural networks is that the distribution of weights in each layer changes between layers, a phenomenon they call *internal covariate shift* [25]. They showed that training can be made more robust by normalizing the inputs to each layer to have zero mean and variance one. More specifically, denote by $x^{(i)}, i \in [1..D]$ the input features to a given layer in the DNN, and by $y^{(i)}$ its output. BN works by normalizing each feature separately, by computing

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mathbb{E}[x^{(i)}]}{\sqrt{\mathrm{Var}[x^{(i)}]}}$$

During training, the estimation of the above mean and variance is done by their batch counterparts. For a minibatch for samples $x_1, \ldots, x_n$ the whole layer is trained by first computing the batch statistics

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$
$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2,$$

Normalizing the inputs to

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

where ϵ is a small constant to avoid division by zero, and then finally scaling the outputs by γ and β, producing the output of the batch normalized layer

$$y = \gamma\hat{x} + \beta.$$

The scaling parameters γ and β are learned for each input feature separately along with the rest of the network, by minimizing the loss function $\mathcal{L}$ for the task at hand. Their function is to allow the network to represent values of *any scale* – not just ones that are zero-centered with unit variance. While the above notation assumes a dense layer with single-vector inputs, the same procedure is easily applied to convolutional layers as well.

The authors also showed that BN allows for larger learning rates, and thus speeds up training. In particular, without it, larger learning rates increase the scale of the weights, which leads to exploding gradients and getting stuck at suboptimal local minima [25]. The need for dropout is also reduced, because BN acts as a regularizer in itself.

Today, almost all state-of-the-art model architectures for various tasks employ batch normalization, and its effectiveness is universally agreed upon [26]. However, why exactly it works has caused conflict among researchers. Recent work by Santurkar et al. shows that reducing internal covariate shift might not in fact be the reason for the technique being effective – the authors show that BN instead makes the optimization landscape smoother, making it easier to optimize [26]. In addition, their experiments show that other normalization strategies that *increase* internal covariate shift yield comparable or better performance.

2.5 Dimensionality reduction and feature learning

High-dimensional data is often difficult to deal with in general, since the larger the dimension of the input data, the larger the computational costs for modelling it become. It is also tricky to visualize data with multiple dimensions, so an important problem in machine learning is how to project the data into a smaller dimension such that it would be both easy to model and plot, and retain as much of the original structure as possible. At the same time, we are also interested in learning features of the data that best describe it. In this chapter we look at common algorithms for dimensionality reduction (DR), as well as unsupervised feature learning with neural networks.

2.5.1 Shallow methods

One of the most basic and popular techniques for dimensionality reduction is principal component analysis (PCA). PCA finds a linear transformation of the data which maximizes the preserved variance, as well as minimizes the mean squared error between the data and its reconstruction [8].

Sometimes it is not enough to transform the data linearly. A non-linear dimensionality reduction method might be able to represent the data is a more interpretable way – for example, for a dataset of hand-written digits, one could imagine to be able to represent the digits as their numeric labels, and such a transformation would certainly not be linear. One of the more popular DR techniques is stochastic neighbour embedding (SNE). SNE attempts to embed the data into a lower dimensional representation in a way that preserves neighborhoods – that is, points that are close to each other in the high-dimensional space should be close to each other in the low-dimensional one as well [27]. The algorithm starts by calculating the probability that the ith sample's neighborhood contains point j, found by assuming a Gaussian distribution of points centered around point i, given by

$$p_{ij} = \frac{e^{-d_{ij}^2}}{\sum_{k \neq j} e^{-d_{ik}^2}}$$

where d_{ij}^2 is a measure of dissimilarity between the points, given by for example the squared euclidean distance between the two points. The *induced probability* that the low-dimensional representation $\boldsymbol{y}_i$ for ith sample has neighbor j is defined as

$$q_{ij} = \frac{e^{-||\boldsymbol{y}_i - \boldsymbol{y}_j||^2}}{\sum_{k \neq j} e^{-||\boldsymbol{y}_i - \boldsymbol{y}_k||^2}}.$$

SNE then minimizes the discrepancy between these two measures by minimizing the Kullback-Leibler divergence between the distributions of neighbors for each point in the two different representations, given by

$$\sum_i \mathrm{KL}(P_i||Q_i) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

Due to the fact that SNE explicitly searches for neighbourhoods of points in the high-dimensional space, it has the nice property of automatically clustering points in the low-dimensional space. This is especially useful for visualizing data for classification tasks where the different classes are easily separable to begin with – a simple SNE plot with colors based on class labels goes a long way in describing the data.

A more popular version of the SNE algorithm is the t-SNE (t-distributed stochastic neighbor embedding) which assumes a heavier-tailed Student-t distribution for the neighborhoods in the low-dimensional space instead of a Gaussian, while also being easier to optimize [28].

2.5.2 Autoencoders

Autoencoders are feature extractors that learn structure from data $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ by first reducing the dimensionality of the data into a low-dimensional space $P < D$ in what is called the encoding phase, and then projecting it back into the original feature space D, called the decoding phase [29]. This procedure loses some detail

in the data, depending on the chosen P, during which the algorithm has to decide which features or combinations of features are most relevant to reconstruct the data as accurately as possible. The idea is then that some deeper structure in the data is revealed by looking at the encoded representation. Autoencoders have been used for example for preserving multi-statement paragraphs [29], speech enchacement [30] and denoising [31].

Autoencoders are a very general framework for feature learning and dimensionality reduction, and indeed many techniques can be represented as autoencoders. For the purpose of this thesis we will only look at autoencoders formulated as neural networks. More specifically, an autoencoder is a neural network that takes as input the data $\boldsymbol{X}$ and produces $\hat{\boldsymbol{X}}$, a reconstruction of the data, and is trained by minimizing the loss (also called the reconstruction error)

$$\arg\min_{\theta} \ \mathcal{L}(\boldsymbol{X}, \hat{\boldsymbol{X}}),$$

where

$$\hat{\boldsymbol{X}} = D(E(\boldsymbol{X})),$$

and E and D are the *encoder* and *decoder*, respectively. We depict E and D as neural networks, so the loss can be efficiently minimized via backpropagation and gradient descent. This allows us to take advantage of the structure of the data, for example by using CNNs when learning from audio or images. A usual choice for the loss function is the mean squared error (MSE), but in the case of images and other structured inputs where the values can be expressed explicitly in the $[0, 1]$ range, binary cross-entropy can also be used, given by

$$\mathcal{L}_{\text{BCE}}(\mathbf{x}, \hat{\mathbf{x}}) = \mathbb{E}_{x \in \mathbf{x}} \left[x \log(\hat{x}) + (1 - x) \log(1 - \hat{x}) \right].$$

An autoencoder constitutes a non-linear dimensionality reduction algorithm if the neural networks that parameterize the encoder and decoder are non-linear in the data. Indeed, a linear autoencoder can be made simply by setting $\boldsymbol{z} = E(\mathbf{x}) = W_1\mathbf{x} + b_1$ and $D(\boldsymbol{z}) = W_2\boldsymbol{z} + b_1$.

2.6 Domain adaptation

In machine learning one tries to fit a model to a particular dataset in a way that maximizes generalization on previously unseen data. However, in many fields of study the unseen data can be fundamentally different from the one being trained on. This phenomenon, known as covariate shift [32] or domain shift [33], means that standard machine learning models trained on the training data do not generalize well to unseen data.

The problem is especially apparent in experimental sciences where experiments need to be performed in the real world. Here a multitude of factors affect the results of an experiment, and correcting such a difference can be difficult to do with conventional means. In particular, in laboratory experiments performed for fouling detection with

ultrasound, the recorded signal can be affected by difficult-to-control parameters such as the temperature of air and how the test setup is built. In addition, signal generators and instruments may provide additional noise that may make the setup sensitive to changes in equipment.

Domain adaptation methods attempt to address these issues by using data from the *target domain* (the one we are interested in generalizing to) to learn features of the *source domain* (the one we have training data for). A usual assumption is then that target domain data is scarce or unlabeled – otherwise we would just train a model on that data.

Domain adaptation is a subclass of transfer learning, and many ways of implementing it have been suggested in literature, such as fine tuning, distribution alignment [33], and domain discrepancy based methods [34]. Recently many *adversarial* DA techniques have been proposed [35, 36, 37], and this chapter focuses on these types of methods.

Adversarial domain adaptation techniques attempt to encode the data in such a way that a classifier trained to distinguish between samples from different datasets cannot do its task properly. This usually involves two competing objectives: maximizing the error that the domain classifier makes while minimizing the error on the *true task* – the underlying problem, be it classification, regression or something else. This approach is motivated by the fact that we wish to base our predictions on features of the data that are *independent of domain*. In a traditional machine learning setting nothing is stopping a model simply trained on source domain data to base its predictions – no matter how good at that task – on features that are exclusive to the source domain. Since we are interested in generalizing to the target domain, it makes sense to explicitly prevent this behavior.

In what follows we cover a broader field of study called *transfer learning*, of which DA is a subfield of. Afterwards, we define concretely the domain adaptation goals and take a look at different techniques in literature.

2.6.1 Transfer learning

In many practical applications we are interested in *transferring knowledge* of a particular task to another task. Indeed, it can be argued that humans transfer skills from other areas of life to be successful at new tasks – for example, when playing a new board game, it helps at understanding the rules to have played similar games before.

Sometimes the term transfer learning is used to describe some sort of fine-tuning, especially in the context of neural networks where weights learned on one task can be adjusted to perform better on another. This thesis, however, treats transfer learning as an umbrella term for techniques that perform different types of knowledge transfer between tasks and datasets.

In the standard supervised learning setting we implicitly assume two things about

the our environment

1. The data that we predict on comes from the same distribution as the one we trained on, $p(X_{test}) = p(X_{train})$
2. The distribution of labels for given data is the same in the training data as it is in the test data, $p(Y|X_{train}) = p(Y|X_{test})$

In other words, we assume that the training and test datasets do not differ too much both in the specific task we are attempting, and in what the data looks like. However, many practical tasks do not conform to these properties. For example, it might be that every time new data is collected, it comes from a new distribution. This is particularly the case in hard sciences where noisy measurement devices and environmental conditions affect the collection of data. In physics, a phenomenon known as the *observer effect* occurs when observing a phenomenon changes it.

Let us look at both properties individually. If the test data comes from a different distribution than the training data, but the task is otherwise the same (for example, in the case of sentiment analysis with book reviews and car reviews) we call this subfield of transfer learning *domain adaptation* – the topic of chapter 2.6. On the other hand, if the second property does not hold, which means that the actual task is different – for example, if we would like to detect zebras and lions based on pictures of cats and dogs – the task is not domain adaptation, since the distribution of the outputs changed between tasks. If neither property holds, we call the task *unsupervised transfer learning.*

A typical use case of transfer learning is found in image detection, where a huge database of images called ImageNet [38] can be used to train models to classify images into thousands of different classes, ranging from animals to objects. If one has a similar image detection task for which not as much data is available, models pre-trained on ImageNet can be used to transfer knowledge from into the other task. This is an example of a problem this thesis does not cover – the labels for the training set are completely different from the ones we wish to predict on.

2.6.2 Problem description

We have established the connection of domain adaptation to more general transfer learning approaches, and defined flexible and efficiently trainable models called neural networks that can be used as building blocks for more complicated future models. In this section we concretely define the domain adaptation problem and present a variety of algorithms for it.

We also cover the two main approaches to domain adaptation: the *supervised* one where labeled target domain data exists but is scarce, and *unsupervised* domain adaptation where no labeled target data is available. Most existing work in the field focuses on the unsupervised version, so it makes for a natural starting point.

Definition 1. *Unsupervised domain adaptation*

Given labeled data from a source domain $\mathcal{D}^S$ consisting of pairs $\boldsymbol{x}_s, y_s$ and target data $\mathcal{D}^T$ with samples $\boldsymbol{x}_t$, find parameters θ for model M_θ that minimize the expected loss on the target domain $\mathcal{D}^T$

$$\arg\min_{\theta} \mathbb{E}_{p(\boldsymbol{x}_t, y_t)} \ \mathcal{L}(M_\theta(\boldsymbol{x}_t), y_t) \tag{6}$$

Note that if the source and target distributions are the same, that is, if $S = T$, then Equation 6 is simply the objective of supervised learning in general, given by Equation 1. Also noteworthy is that we do not define any specific true task in our objective, just as long as we have a model with a set of unknown parameters θ and a loss function $\mathcal{L}$ that tells us how much error we are making, the true task can be any *supervised* learning objective, such as classification or regression. In fact, in this work we present an algorithm that learns a regression problem with almost no supervision, and apply domain adaptation to that model.

We can also interpret the above loss as trying to minimize the discrepancy between the two domains. If we were to learn a mapping between the domains, we could simply map target domain samples into the source domain space and apply our model there. This is indeed the objective of adversarial domain adaptation, the subject of the next section. There exist ways to do domain adaptation that do not rely on neural networks or adversarial learning, but we do not cover them in this thesis.

2.6.3 Adversarial models

In 2014 Goodfellow et al. [39] showed how a pair of neural networks competing with each other could be used to create a model that generates convincing-looking images. This approach, called generative adversarial networks (GANs), gave birth to successful applications of similar techniques in many fields. While GANs are used to generate data, adversarial learning has also been successfully applied to domain adaptation as well. This chapter delves into the theory of adversarial learning, as well as its applications in domain adaptation.

The more general way of training generative models usually involves making assumptions about the distribution of data, and sampling from such distributions to infer their parameters. This process can be highly time-consuming since analytically the derivatives of such models do not behave well, and a large amount of sampling may be required to approximate the distributions.

Goodfellow et al. show how a generative model can be trained without doing these types of intractable computations. They use a different approach where the generative model is faced with an adversary whose goal is to tell apart the data sampled from the model and the real data it is trying to model. To perform well, the generative model, called the *generator* needs to produce convincing enough samples

that the *discriminator* cannot tell them apart from real data. Both the generator and discriminator are modeled as neural networks – therefore they can be trained efficiently via gradient descent and backpropagation.

In adversarial learning the two competing objectives are trained concurrently. This corresponds to two players playing a zero-sum (their gains and losses are in exact balance) minimax game (minimizing the loss in the worst case) [39]. More specifically, the generator G tries to map noise $\mathbf{z}$ into something that is as close to real data as possible, while the discriminator D tries to tell apart the real data from the fake data.

Given the generator G and discriminator D, the learning objective of GANs can be described as the following (eq. 1 in [39])

$$\min_G \max_D \; \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})} \left[\log D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})} \left[\log(1 - D(G(\mathbf{z})))\right]$$

where $\mathbf{z}$ is a vector of noise fed into the generator to produce realistic looking data. This kind of an objective can be trained efficiently in modern deep learning frameworks via gradient descent. Very similar objectives will be seen in the next chapter that use the same ideas and concepts as GANs, even if they do not explicitly generate data.

In practice, in the early stages of training, it is easy for the discriminator to tell apart generated and real data, so it converges quickly. This causes the gradient for $\log(1-D(G(\mathbf{z})))$ to vanish [39]. Therefore, instead of minimizing the aforementioned objective, it is common to instead maximize $\log(D(G(\mathbf{z}))$.

2.6.4 Adversarial domain adaptation

If the source and target domains are sufficiently different from each other, it is usually easy to train a model to distinguish between them. This is a problem because any model trying to accomplish the true task on the source domain might rely on features of the data that are exclusive to the source domain, and thus be rendered useless on the target domain. One idea then, is to try to encode the source and target data points into such a format that it is impossible to distinguish between the domains, and then use this common ground as data for our true task.

The key here is that if we can indeed represent data points from both domains in this way, any classifier that does well on source data should also do well on the target domain.

Just as we saw with GANs, the idea that arises is to train a classifier – which we again call the *discriminator* – that given an encoded data point tries to classify the domain where it came from. If it were possible to fool the discriminator by representing the data differently, we could be confident that the model worked well on both domains.

More formally, an encoded representation of a data point $x \in \mathbb{R}^p$ is any function $f(x) \to \mathbb{R}^e$ where usually $e < p$. The equations for learning such a mapping that

enforces domain invariance is then given by Definition 2.

Definition 2. *Adversarial unsupervised domain adaptation*

Given data from a source domain $\mathcal{D}^S$ consisting of pairs $\boldsymbol{x}_s, y_s$ and target data $\mathcal{D}^T$ with samples $\boldsymbol{x}_t$, find parameters θ for encoder $E_\theta(\boldsymbol{x})$ such that

1. *$E_\theta(\boldsymbol{x})$ minimizes the expected loss on the true task on the target domain*

$$\arg\min_\theta \ \mathbb{E}_{p(\boldsymbol{x}_t, y_t)}[\ \mathcal{L}(M(E_\theta(\boldsymbol{x}_t)), y_t)\ ] \tag{7}$$

2. *Given an instance $\boldsymbol{x}_d$, $d \in \{s, t\}$ from one of the sources, and a domain discriminator $D(E_\theta(\boldsymbol{x}_t)) \to d$, D cannot reliably distinguish the domain*

$$\arg\max_\theta \ \mathbb{E}_{p(\boldsymbol{x}, d)}[\ \mathcal{L}(D(E_\theta(\boldsymbol{x}), d))\ ] \tag{8}$$

Both of the objectives are equally important, because without (7) E can just map all samples into noise, and without (8) our model is equivalent to training a classifier on the source domain and applying it on the target domain. The loss function for Eq. 7 can be any supervised loss, but Eq. 8 corresponds to a binary classification problem, so either $\mathcal{L}_{\text{BCE}}$ (3) or $\mathcal{L}_{\text{CE}}$ (5) should be used.

As mentioned before, since deep learning models are very flexible models, it is common to represent both the encoder and discriminator as neural networks, but other formulations are also possible. Using neural networks has the additional benefit of being able to optimize both objectives concurrently by optimizing their sum.

2.6.5 Overview of literature

Different adversarial domain adaptation algorithms exist in literature. This chapter looks into different approaches to provide an overview of the field. We only consider classification tasks in this chapter, but most algorithms generalize to regression problems as well.

In [40], one of the first papers to use adversarial training for domain adaptation, a domain discriminator is concurrently learned with the main objective, and used as a regularizer. As a result, it is easy to tune the amount of effect DA has on the final model, with the regularization strength parameter $\lambda = 0$ being equal to training on the source domain and evaluating on the test domain.

The model uses a so-called "gradient-reversal layer" (GR) which essentially flips the sign of the gradient during backpropagation, resulting in the objective being maximized for the layers affected, i.e the ones after the GR layer. This trick makes

it easy to implement the network in modern deep learning libraries, but is equivalent to maximizing a loss function w.r.t a subset of weights and minimizing for the rest.

On the other hand, Tzeng et al. [35] use a two-step approach where a high-quality model for the main task is first learned from only the source data. The idea is to learn a completely separate encoder for the target data, which is made similar to the source encoder by using an adversarial loss. Their algorithm is called Adversarial Discriminative Domain Adaptation (ADDA), and is structured as follows.

1. Learn classifier and source encoder on labeled source data.
2. Learn target encoder and domain discriminator by freezing the source encoder and training target encoder to fool the discriminator.
3. During evaluation, the only the target encoder and classifier are used.

More specifically, denote by E_s, E_t the source and target encoders, respectively, by M the classifier (which is used for both domains), and by D the domain discriminator. Then, given the source samples X_s and labels Y_s, step 1 first optimizes the standard cross-entropy loss (eq. 5) on the source data:

$$\min_{E_s} \mathbb{E}_{\mathbf{x}_s, y_s \sim (X_s, Y_s)} \left[\sum_{c}^{C} \mathbb{1}[c = y_s] \log(M(\mathbf{x}_s)) \right] \tag{9}$$

This formulation only considers classification as the main task, but extension to regression problems is straightforward.

Next, in the domain adaptation step we jointly optimize the following objectives that correspond to domain confusion

$$\min_{D} -\mathbb{E}_{\mathbf{x}_s \sim X_s}[\log D(E_s(\mathbf{x}_s))] - \mathbb{E}_{\mathbf{x}_t \sim X_t}[\log 1 - D(E_t(\mathbf{x}_t))] \tag{10}$$

$$\min_{E_t} -\mathbb{E}_{\mathbf{x}_t \sim X_t}[\log D(E_t(\mathbf{x}_t))] \tag{11}$$

These objectives yield step 2 of the algorithm. They correspond to optimizing the domain discriminator to be able to separate the domains and optimizing the target encoder to fool the discriminator, respectively.

During evaluation, E_s and D can be thrown away, and predictions for instances of the target domain $\mathbf{x}_t$ are given by

$$\hat{y} = \arg\max_{c} M(E_t(\mathbf{x}_t)).$$

The authors use a standard CNN for image tasks as the encoder structure, along with a shallow dense network as the classifier. It is common for authors to benchmark

their results on not only image classification tasks, but also text datasets, such as reviews, where reviews of different categories of items often exhibit a domain shift. These tasks are generally solved using deep learning, so evaluating on them makes it is easy to compare to non-DA methods.

In addition to unsupervised approaches, supervised domain adaptation is also a prominent research area. In this case we usually assume that the amount of labeled target data is very scarce, and we might have as few as a single sample per class.

Motiian et al. [37] suggest a method for supervised domain adaptation using just a few samples of labeled target domain data. Instead of training a discriminator that can discriminate between two domains, they introduce the Domain-Class Discriminator (DCD) that takes in a concatenated vector of two samples, and classifies the samples into one of four classes:

1. Same class and domain
2. Same class but different domain
3. Different class but same domain
4. Different class and different domain

Importantly, the DCD should only confuse groups 1 and 2, and groups 3 and 4 with each other. This is because for example classifying instances of group 1 as group 4 would mean that the model cannot distinguish between instances of *different class*, which is detrimental to the main task. The authors postulate that this additional separation introduces *semantic alignment* which simultaneously affects our two goals: domain confusion and class separability.

3 Ultrasonic fouling detection

3.1 Problem statement

Since industrial processes that develop fouling can be significantly hampered by the buildup, cleaning them can be expensive. For example, the method for cleaning heat exchangers might include halting production and applying cleaning agent, which can halt the process for days. Ideally, we would like to know when cleaning is required, and only perform such an expensive process if it is truly needed. However, for example in the case of pipes, while fouling can be easily detected by halting the process and having a look inside, we would like to avoid unnecessary stops in production.

One method for detecting fouling without intervention is to emit ultrasound from outside into the equipment and measuring the signal that comes back [1]. The idea is that when fouling is present, multiple physical properties of the signal change with respect to the clean state. Some of these include: (1) the echo of the signal is weakened, (2) the time of flight is shorter due to the signal hitting a substance earlier, (3) the acoustic impedance at the boundary of fouling changes.

Recent work by Altum Technologies [2] has shown that it is also possible to *clean* the equipment using ultrasound, in a completely non-invasive manner. This kind of technology presents a huge step forward in terms of cleaning industrial fouling without stopping producing, resulting in massive savings of both monetary and ecological value [41]. Non-invasive fouling detection in this case becomes even more important, since equipment that is cleaned by ultrasound would still need to be opened and checked for fouling every once in a while without non-invasive detection of fouling.

From a machine learning perspective, one could simply start solving this problem as a case of supervised learning. However, we are faced with multiple difficulties: first, it is difficult to obtain labeled training samples since it is not possible, in general, to look inside the equipment while cleaning is taking place. Secondly, industrial equipment is rarely exactly the same in terms of the shape of the pipe etc, or the materials used in building it – instead, there is great variance in for example the shapes and sizes of heat exchangers and pipes. Also the exact composition of the fouling substance itself is often difficult to find out, as it can be a combination of different agents and byproducts of the industrial process that is being run to produce goods. These points mean that it is probable that the data-yielding distribution $P(X)$ changes between the training and test scenarios, and techniques presented in Section 2.6 are required.

Thirdly, ultrasonic reflections can be noisy, especially in the case where something is being produced while we are attempting to detect fouling. Artefacts and noise are in this case to be expected. Finally, the high-dimensional signal produced is non-trivial to use as input for a machine learning model.

Wallhauser et al. [42] showed that by using knowledge of the materials the fouling

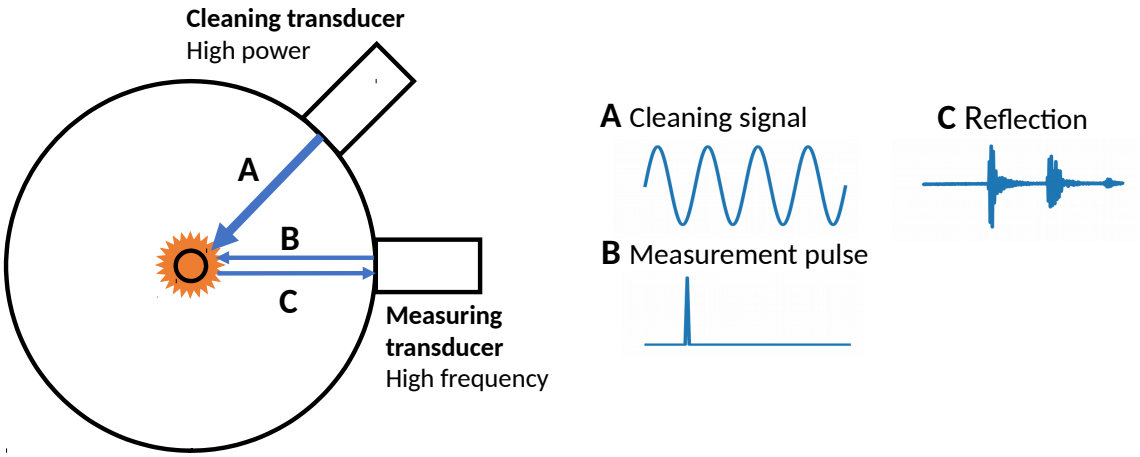

Figure 7: Overview of the ultrasonic cleaning and fouling detection setup. The orange material represents the fouling attached to an internal pipe. High power ultrasound (A) is used to clean the pipe along with high frequency measurement pulses (B), which are used for detecting fouling using their echoes (C). A and B are interleaved, so we are able to detect fouling while cleaning is running.

and pipes consist of, they can calculate physical properties such as time-of-flight of the signal and feed these features into an off-the-shelf neural network that performs well on their particular setup. However, were the type of fouling, equipment or something else about the setup change, then it would be difficult for the same model to generalize to new data because the features would need to be recalculated.

In this work we are only interested in models that require no a priori knowledge about the type of fouling or properties of the industrial process, and work with raw ultrasound signal only. This is because solutions that rely on specific properties of equipment require extensive recalculation when applied to new equipment – in some cases the calculation might even be impractical to begin with.

3.2 Experimental setup

To evaluate the effectiveness of the presented algorithms we look at two different scenarios of fouling detection:

(a) Detecting fouling in a single run after the run is complete.

(b) Detecting, in real time, when the equipment is clean based on training samples obtained from previous runs.

Even though (b) provides more value in the sense that the cleaning process can be stopped when the equipment is clean, saving energy, (a) can also be useful for providing a rough estimate of how long cleaning typically takes on a given equipment. (a) is also easier to attain, since it does not require multiple runs from fouled to clean, which may potentially take a long time depending on the equipment.

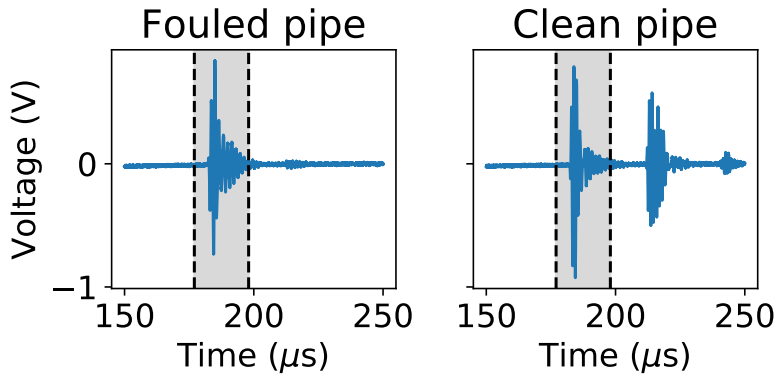

Figure 8: Example measurements for fouled (left) and clean (right) structure. The shaded area indicates the part of the signal used as model input – the model does not see the second surface echo, which we use for evaluation.

Our experimental setup consists of an acrylic pipe with a single internal structure that is fouled with quicklime. The pipe is equipped with a high-intensity transducer which cleans the pipe using proprietary technology provided by Altum Technologies, each pulse of which is interleaved with a pulse from a secondary high-frequency measurement transducer. This way we can clean the pipe while recording signals sufficiently often. When measuring, the measurement transducer in run in pulse echo-mode, where a delta spike is emitted, and its echoes are saved. Measurement pulses are saved as often as the oscilloscope can save them to disk, which amounts to approximately 0.1 seconds per pulse, or 10 times a second. Figure 7 shows a diagram of the setup.

Since our setup is made from acrylic, the measurement signals pass through it, which would not happen in equipment made from, for example, steel. As such, the measurement signal contains two main echos, one for each water-acrylic surface in the inner pipe. This allows us to evaluate the effectiveness of our approaches by looking at the amplitude of the secondary echo; fouling must be present in the pipe if the second echo is weak. Figure 8 gives an example of these two echos.

We perform each run from completely fouled to completely clean, and keep the experiment running to make sure that a sufficient amount of samples at the end are clean. Seven such runs were made, with varying lengths due to variance in the amount of fouling, with the average length being around 1.6 minutes. In some runs the inner pipe was cleaned very quickly, in a matter of seconds, and in others it took a few minutes. Some of the runs exhibit a pulsating effect, in that the amplitude between sequential samples seems to fluctuate. One possible explanation for this is that the cavitation that forms due to the ultrasonic cleaning lingers long enough for it to effect the measurement signal.

Each example in our dataset is a signal of 100 μs for which we have 10000 samples. The signals are captured after 150 μs have passed from the start of the measurement pulse. We clip the pulses so that the a window that covers the first echo is used for

training, and the second echo is used for evaluation.

One of the seven runs was additionally discarded due to it being already clean according to the second echo. It is likely that the fouling in this case was so unevenly spread that the ultrasonic pulse could not detect it.

Our objectives in the two detection scenarios are as follows. In (a) we are given a time series of signals $X \in \mathbb{R}^{N \times D}$ captured using the interleaved cleaning and measuring setup. We assume that when cleaning began, the equipment was fouled, and when cleaning stopped, that is, at the end of our measurement, the equipment is completely clean. The length of the run N, may vary depending on how quickly the equipment got cleaned. We would like to know at which point in this time series the equipment was actually clean – that is, find the point $p \leq N$ at which cleaning had already taken place.

In (b) on the other hand, we have a set of previous experiments $X_1, X_2 \dots X_m$, each consisting of a time series of signals $X_m \in \mathbb{R}^{N \times D}$ captured during cleaning. After training the model on the historical data, given a new measurement signal we wish to predict whether or not it is fouled, that is, give a binary prediction $\hat{y} \in [0, 1]$.

3.3 Data exploration

Before running any experiments, we investigate the dataset by reducing its dimensionality so we can plot each signal in a two-dimensional plot. This allows us to see if there are any patterns related to the two classes or the different experiments. To do this, we train a convolutional autoencoder (Section 2.5.2) on the entire dataset, and apply the PCA algorithm (Section 2.5.1 on the encoded representation to visualize the data. More specifically, the autoencoder used consists of 3 layers temporal convolution, each with batch normalization (Section 2.4.8) and ReLU-activations as the encoder, and 3 layers of transposed convolution [21], again with batch normalization and ReLU activations as the decoder. Transposed convolution increases the size of its input spatially, allowing us to reconstruct the signal to its original dimension.

The encoder compresses the dimension of the input from 1000 to 154, from which the decoder reconstructs it. After training, we only keep the encoder and use it to compress the entire dataset, after which we apply the PCA algorithm. Figure 9 gives the results for this visualization. Each color corresponds to a single experiment, and each point to a single signal in the data.

It is apparent from Figure 9 that the separate runs are fundamentally different, and clearly separable using a simple classifier. The reason for this domain shift could be a combination of factors, such as the unevenness in the spread of fouling in the inner pipe, temperature differences, differences in attachment of the inner acrylic pipes, and so forth. In contrast to the traditional domain adaptation setting however, if we were to train a model to learn features from which the different domains could not be told apart, an unfortunate consequence would be that we would be telling the model to also ignore any possible differences between fouled instances as well.

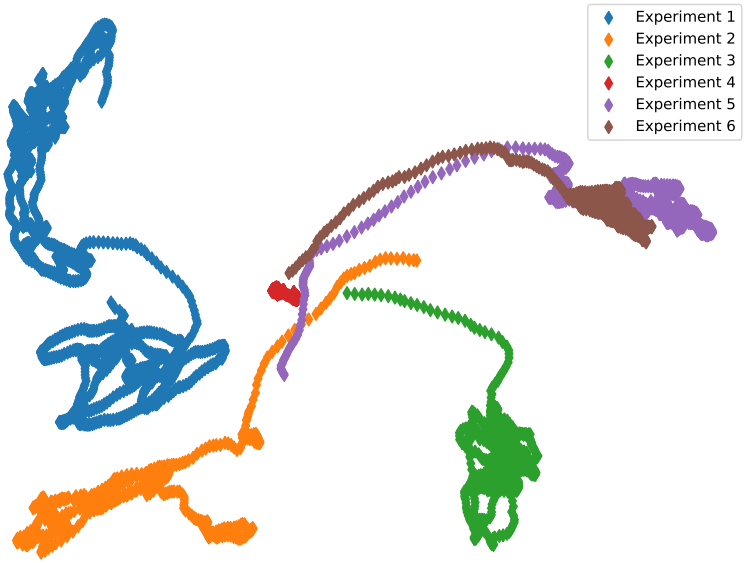

Figure 9: Each sample in the dataset encoded into the latent representation of a convolutional autoencoder, and then projected onto the first two principal components using the PCA algorithm. Note how the different experiments are easily separable – this indicates a domain shift between them.

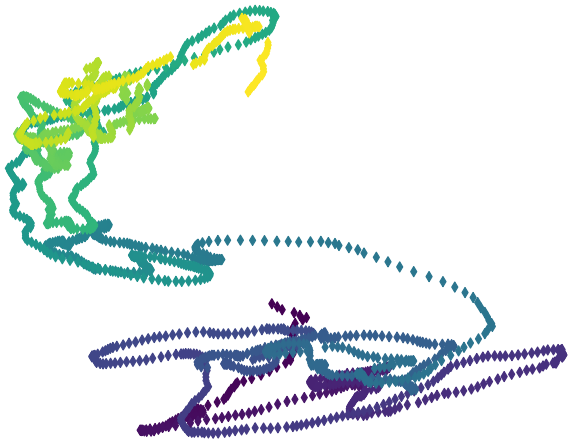

Figure 10: Experiment 1 from Figure 9 zoomed in and recoloured. Colour indicates position in time series: lighter samples were captured early during cleaning, and darker ones later. Note how there is a clear temporal separation into two parts, clean and fouled.

The problem is that different types and amounts of fouling *do* present differently in the ultrasound echo, and thus it does not make sense to force our model to put differently fouled samples in the same category.

Instead, we would like for our model to not be able to tell apart the *clean samples from different runs*, while preserving the fact that differently fouled samples will be distinguishable from each other. For the fouling detection experiments we noted that a simple model was able to generalize between the experiments, and further domain adaptation for not necessary for achieving good accuracy. The domain shift problem might become more apparent in new experiments, so we explore solving a similar task using a toy dataset in Chapter 5.

We can also use the same autoencoder to look at the individual experiments. In Figure 10 we show that within some of the experiments there is a clear difference between clean and fouled samples in the low-dimensional representations. Such a clear separation indicates that this problem is solvable with very little labeled data.

4 Semi-supervised fouling detection

In this chapter we present the problem of semi-supervised fouling detection from the perspective of machine learning, and introduce algorithms for detecting the amount of fouling in ultrasonic cleaning setups when labeled data is not available for each collected sample. Instead, we rely on assumptions about the cleaning process, namely, that it is monotonic in the sense that the amount of fouling can never increase as cleaning is continued. In previous work by Longi et al. [5], the pseudolabel approach presented in Section 2.3 is extended by showing the so-called *structured pseudolabel* technique. We further extend it by applying the monotonicity constraint to it.

In addition, a CNN architecture specifically crafted for the task is presented. The model builds upon previous work by Phan et al. [43] on CNNs for audio event recognition, and encodes high temporal invariance and robustness into the model itself. The model is simple enough to generalize well, but complex enough to capture the intricacies of high-dimensional audio signal.

4.1 CNNs for fouling detection

Since we are interested in learning from high dimensional data with almost no supervision, it makes sense to take into account as much a priori information about the task at hand as possible, and encode it into the model. By thinking about *what kind* of signals we are specifically dealing with, it is possible to create model architectures that work well this task, as demonstrated using the toy data example in Section 2.4.6.

More specifically, as demonstrated in Figure 8, the input signals to our model contain, by their physical properties, several distinct spikes – one for each surface the ultrasonic signal has to go through when travelling back to the transmitter. Fouled equipment also tends to affect the time-of-flight of the measurement signal – in other words, the exact temporal location of the spikes might be altered. In between those spikes there is generally only low-amplitude noise, since the sound simply goes straight through the medium, which in our experiments is water. See Figure 8 for two such signals, one taken from a fouled pipe in our experimental setup and one from a clean one.

Given these assertions about the input data, we can build the CNN architecture in a way that corresponds to our prior knowledge about the phenomenon. We begin by noting that since the exact location of each spike might vary based on the internal structure as well as amount of fouling, the model needs to be highly invariant temporally. This property is often associated with the pooling operation.

In previous literature, Phan et al. [43] showed how to use a particular kind of pooling that reduces – in the temporal convolution case – the entire feature map of a filter into a single scalar, called *1-max pooling*, for audio event recognition. Pooling over the entire set of activations enforces high temporal invariance of features, since it essentially makes the location of the spikes irrelevant. Since this pooling operation

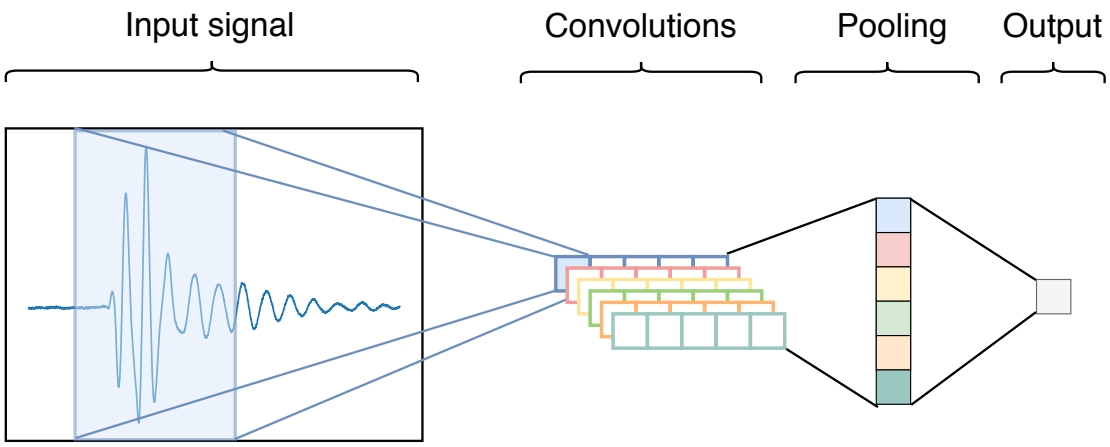

Figure 11: The CNN model architecture used for fouling detection. A total of 16 filters are learned using 1-max pooling which reduces the entire feature map to a single scalar. The input is a raw signal, and the output is the probability that fouling is present.

also greatly reduces the amount of parameters in the model, it could also lead to better generalisation.

It should also be noted that the reduction in time-of-flight (ToF) due to fouling is a small quantity easily covered by the filter size, so a single activation of a filter of sufficient size should be enough to detect the phenomenon.

The final model used for our experiments is given by Figure 11. It consists of 16 ReLU-activated filters of size 5, followed by one-max pooling, which then connects to a dense layer with 16 inputs and one output: the probability of fouling. We further apply a sigmoid activation to the last layer in order to keep the outputs of the model in the interval $[0, 1]$. For training, the Adam optimizer is used, with an initial learning rate of 0.001.

The parameters were chosen via grid search using cross-validation. In particular, the amount of filters were chosen by noting that of the four values tried (8, 16, 32 and 64), no single value provided a significant increase in accuracy. More complex models with more parameters in the form of layers and filters tended to overfit very quickly, so a less complex model was employed instead.

4.2 Monotonized pseudolabels

4.2.1 Label monotonicity

During the collection of training data, the runs can be structured so that we are certain that in the beginning the pipe is fouled, and at the end it is clean. This gives us labels for some unknown amount of samples from each end of each run, but the ones in the middle are left as unlabeled. One way to use the unlabeled data is by applying the pseudolabel technique presented in Section 2.3. However, if a mistake is

made by falsely labeling a clean datapoint as fouled, the mistake will be emphasized in later iterations of the algorithm, and will result in subpar performance.

Given the fact that the amount of fouling can never increase during a training run, we should enforce this property in the pseudolabeling step. In the application of identifying wi-fi interference, Longi et al. [5] encountered a similar problem where it is not likely that a device would intermittently cause interference (i.e be on), and suddenly disappear only to reappear in a few milliseconds. To address this issue, they introduced *structured pseudo-labels* that apply a penalty for each time temporally consecutive samples are given the same label. Denote by $\hat{y}_i$ the pseudolabel for the ith sample. The additive term to the loss that enforces temporal continuity in labels is then given by

$$\lambda \sum_{i=2}^{n} \mathbb{1}[\hat{y}_{i-1} \neq \hat{y}_i]$$

where λ is a hyperparameter that controls the scale of the penalty that occurs for each time consecutive labels get assigned different labels.

Longi et al. [5] further use the Viterbi algorithm for Hidden Markov Models [8] to find the optimal sequence of assignments given a sequence of predicted pseudolabels. In the task they study, devices can turn off and come back on at a later time. However, in the case of fouling detection, we assume that during training once the equipment is clean, it cannot develop more fouling. Therefore, a simpler algorithm can be used to assign the monotonic pseudolabels.

Assume that for a time series of N measurements, the equipment is at first fouled (labeled 1) and at time step t becomes clean, after which it continues to be clean until the end of the run. Our task is to assign pseudolabels $\hat{y}_i$ to unlabeled samples so that for some timestep $t \leq N$ each pseudolabel $\hat{y}_i = 1 \forall i < t$ and $\hat{y}_i = 0, \forall i > t$. There are N total samples, of which we denote by N_l the amount of labeled ones and by N_u the amount of unlabeled ones.

Denote by z_i the predicted probability that the ith unlabeled sample is fouled, given by the CNN model. The objective then corresponds to the following optimization problem:

$$\arg\max_{t} \left[\sum_{i=1}^{t} \log \hat{z}_i + \sum_{i=t+1}^{N_u} \log(1 - z_i) \right] \tag{12}$$

where n is the last unlabeled sample. The optimal t is found by iterating over all possible ts, computing the given quantity, and picking the maximum value. An outline of this algorithm is given by Algorithm 2.

4.2.2 Probability monotonicity

The approach just presented, which we call *label monotonicity*, works well in practice, as shown by our experiments in Section 5. The implicit assumption that is being

Algorithm 2 Label Monotonicity supervision for neural networks

1: Initialize network parameters θ for randomly
2: Train network on $N_l = 2k$ labeled instances
3: Predict labels $\hat{\boldsymbol{z}}$ for all N_u unlabeled samples
4: **for** i in iters **do**
5: Set $\hat{\boldsymbol{y}}$ by solving

$$\arg\max_{t} \left[\sum_{i=1}^{t} \log \hat{z}_i + \sum_{i=t+1}^{N_u} \log(1 - z_i) \right] \quad (12)$$

6: Update $\boldsymbol{\theta}$ to improve (12) plus

$$- \sum_{i=1}^{N_u+N_l} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \quad (3)$$

made here is that during cleaning, the fouling comes off as a large chunk, which proved to be the case for our laboratory setup. However, for more persistent fouling that might take longer to clean, it is plausible to think that if it comes off small piece by piece, the model could capture this by predicting a smaller probability of being fouled.

For such a scenario we present another method, called *probability monotonicity*, which instead of giving hard assignments as pseudolabels learns the probability of the equipment being fouled, or more intuitively, the amount of fouling. This method assumes that the probability of the equipment being fouled decreases monotonously, and assigns continuous pseudolabels $\hat{z}_i \in [0, 1]$ for each unlabeled sample, given by solving the following constraint optimization problem:

$$\begin{aligned} \arg\min_{\hat{\boldsymbol{z}}} & \sum_{i=1}^{N_u} (z_i - \hat{z}_i)^2 \\ \text{s.t.} \quad & \hat{z}_i - \hat{z}_{i-1} \leq 0 \quad \forall i \in [2, \ldots, N_u], \end{aligned} \quad (13)$$

The objective corresponds to a set of labels where consecutive labels are monotonically decreasing in the probability of fouling, and match the predicted labels as closely as possible. To learn an optimal solution for (13), we note that the objective is that of isotonic regression [44], which can be efficiently solved using the Pool-Adjacent Violators Algorithm (PAVA) [45].

The probability monotonicity algorithm is outlined in Algorithm 3. It is essentially the same as Algorithm 2, but the PAVA algorithm is used in the pseudolabel assignment step, and the loss function used for the pseudo-labels is the mean squared error (2) instead of binary cross-entropy (3). For the fixed labels we use binary

Algorithm 3 Probability Monotonicity supervision for neural networks

1: Initialize network parameters θ for randomly
2: Train network on $N_l = 2k$ labeled instances
3: Predict labels $\hat{\boldsymbol{z}}$ for all N_u unlabeled samples
4: **for** i in iters **do**
5: Set $\hat{\boldsymbol{z}}$ by solving using the PAVA algorithm

$$\arg\min_{\hat{\boldsymbol{z}}} \sum_{i=1}^{N_u} (z_i - \hat{z}_i)^2$$
$$\text{s.t.} \quad \hat{z}_i - \hat{z}_{i-1} \leq 0 \quad \forall i \in [2, \ldots, N_u] \quad (13)$$

6: Update $\boldsymbol{\theta}$ to improve (13) plus

$$\frac{1}{N_l + N_u} \sum_{i=1}^{N_l+N_u} (\hat{y}_i - y_i)^2 \quad (2)$$

cross-entropy given by equation 3. Refer to Figure 12 in Section 5 for a comparison of the effect of both variants of monotonicity constraints on predictions.

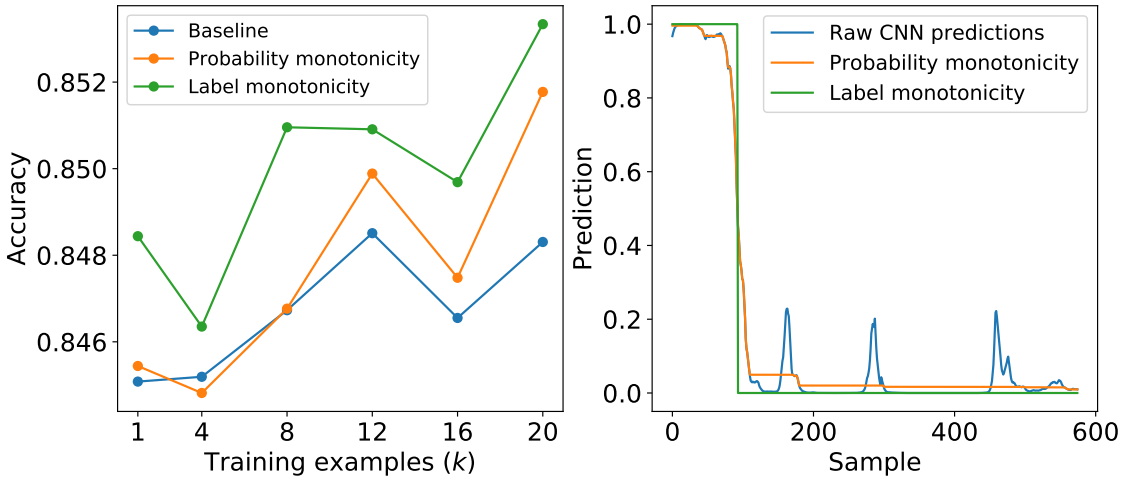

Figure 12: Left: the mean accuracy of detection for the single-measurement case. In this case all the variants perform very similarly, although the label monotonicity variant is the best by a small margin. Right: the effect of enforcing monotonicity using both presented variants. Note how the pseudo-label approaches nullify the erroneous spikes in the raw CNN predictions.

5 Experimental results

In this chapter we cover the results of our experiments. The first two experiments are conducted with the setup described in Section 3.2, while the adversarial adaptation demonstration in Section 5.3 is based on a generated artificial dataset.

5.1 Fouling detection after the fact

The problem is approached as follows. We pick k samples from each end of the dataset as fouled and clean, respectively. The fouling detection CNN is then trained using these samples, and pseudolabeling is performed, alternating between optimizing the model parameters and pseudolabels, as outlined in algorithms 2 and 3. For each run we measure the mean accuracy for different choices of k, given by

$$\frac{1}{n}\sum_{i}^{n} y_i z_i + (1-y_i)(1-z_i).$$

The left part of Figure 12 gives the accuracy for this experiment, as a function of k, the amount of samples chosen from each end. As noted, the label monotonicity variant marginally outperforms the other variants, but they all solve the task adequately. The right part of the figure displays original predictions of the CNN in blue, where one can see three false positive spikes in the probability of fouling. In this case the mistakes made by the classifier were not enough to its predictions radically, but in principle enforcing monotonicity should root out such errors.

These results show that it is indeed possible to detect the point at which the equipment was clean after a single run from fouled to clean using our experimental setup, with high accuracy. In particular, the CNN architecture presented in this work seems to work well for this task, since the algorithm of choice did not play a large role in the accuracy in Figure 12. A notable property of the presented pseudo-label approaches is that they do not, overall, reduce accuracy. The right part of figure 12 also shows that even though the mistakes in blue were not able to influence the raw CNN predictions in this case, the pseudo-label algorithms get rid of them entirely.

It is also evident that only a few labels from each end, e.g $k = 1$ is enough to solve the task. This again shows how robust the simple CNN model is – it is rare for neural networks to generalize so well with so little data.

5.2 Real-time fouling detection

To simulate the detection of fouling in real time, we perform leave-one-experiment-out cross-validation by training the presented methods using five of the six available runs, and evaluating on the final one.

Figure 13 gives the results for the real-time detection task. The leftmost plot shows the accuracy of all variants of pseudolabeling, including the one that does not enforce monotonicity. In this case enforcing monotonicity does not help with overall accuracy. The center and rightmost plots give example predictions for two runs. While the center run is predicted mostly correctly using all methods, it can be seen from the rightmost plot that both monotonicity variants help reduce the erroneous spike in the predictions at around sample 300. Especially poor is the baseline CNN which would change its binary label prediction for said spike.

Another baseline, logistic regression, was also tested on the dataset, but the results were poor – the algorithm amounted to essentially guessing, as the accuracy was as good as guessing the most common label on the training set.

Based on these results we can say that also the real-time detection task is solvable with the given techniques. In this case the pseudo-label approaches offer a better-than-marginal improvement across the board, although the presented monotonicity-enforcing variants are no better than regular pseudo-label on our experiments. It is also worth noting that this task is also solvable using just one labeled sample from each end.

With regards to the domain shift phenomena that was shown to be present in Chapter 3, the results show that domain adaptation is not required to solve this task in laboratory conditions. Again, the CNN model used is so simple that it is likely robust to the domain shift in this case. Therefore we investigate the use of domain adaptation methods on toy data, presented in the next section.

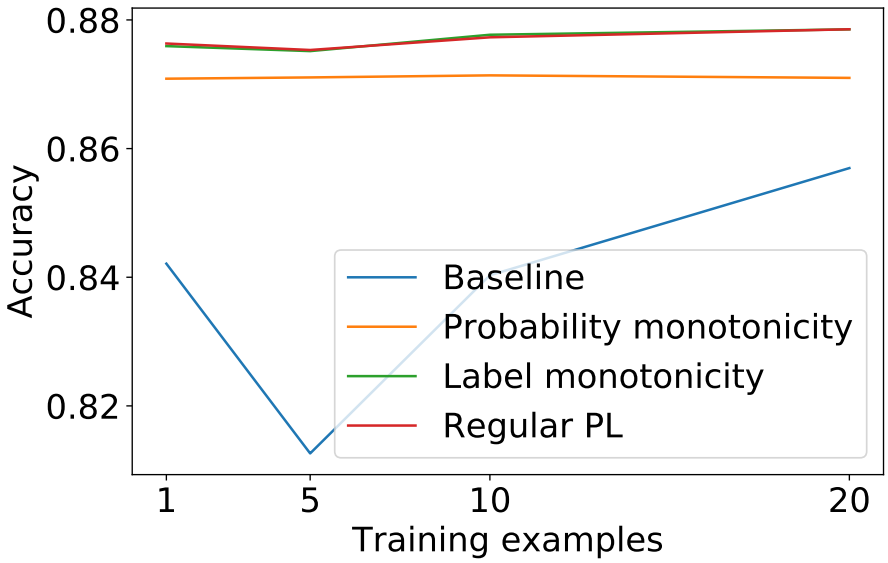

Figure 13: The effectiveness of the two pseudo-label variants and the baseline on the real-time detection task. Both pseudo-label techniques outperform the baseline. The vertical axis denotes k, the amount of samples used from both ends as labeled. All pseudolabel methods perform better than the baseline, including the standard one with no monotonicity constraints.

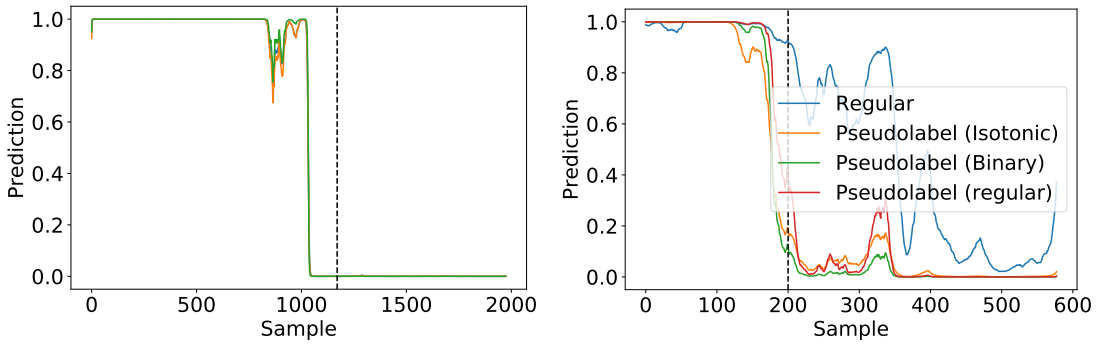

Figure 14: Two examples of runs, one where the pseudo-label methods match the baseline and one where the baseline is significantly worse. The black vertical line gives the ground-truth using the second surface echo. Note that the regular PL (red) is as good as the label monotonicity variant (green).

5.3 Adversarial adaptation

After evaluating domain adaptation on the laboratory data, it was found that it did nothing to help accuracy. This is likely because the CNN model has so few parameters to begin with, that it has to focus on more or less domain invariant features anyway. Regardless, we would like to evaluate the effectiveness of adversarial

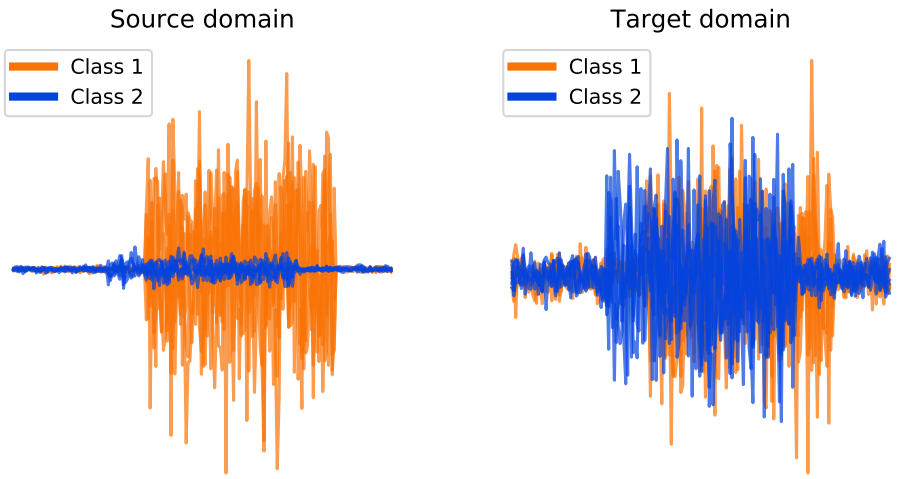

Figure 15: Left: five signals per class sampled from the source distribution of the toydata problem. Right: five signals per class from the target distribution. Both distributions exhibit the temporal shift between classes, but the source domain also has a scale discrepancy between classes. It is easy to solve the source task by just using the scale difference, but such an approach would not generalize to the target domain.

domain adaptation in the context of signal data, so we perform an experiment on toy data instead.

A binary classification task with two domains, a source and a target domain, is generated by creating two distributions of "signals" of $d = 200$ features. Figure 15 shows examples sampled from both domains and both classes. Both distributions contain a noise part and a *spike*, where the class 1 spike always starts earlier in the signal. In the source distribution the amplitude of the spike in instances of class 1 is significantly higher than those of class 2, that is, the absolute value of the spike is higher. This dataset simulates a situation where the amplitude of signals varies between the source and target distributions, but the time of flight does not.

As is evident in the experiments, it is easy for a classifier trained only on the source domain to only learn the amplitude difference, achieving perfect accuracy on that domain. However, the task can be solved in both domains by simply looking at where the spike begins.

Experiments are performed on this dataset by first sampling $n = 1000$ samples from the source domain, half of which belong to class 1 and the other half to class 2. We then apply the unsupervised DA model shown by Tzeng et al. [35] as presented in Section 2.6. This model makes it easy to compare DA against the baseline of just learning on the source dataset, since the technique involves a pretraining step on the source domain.

As the encoder we use a simple CNN consisting of a single ReLU-activated temporal

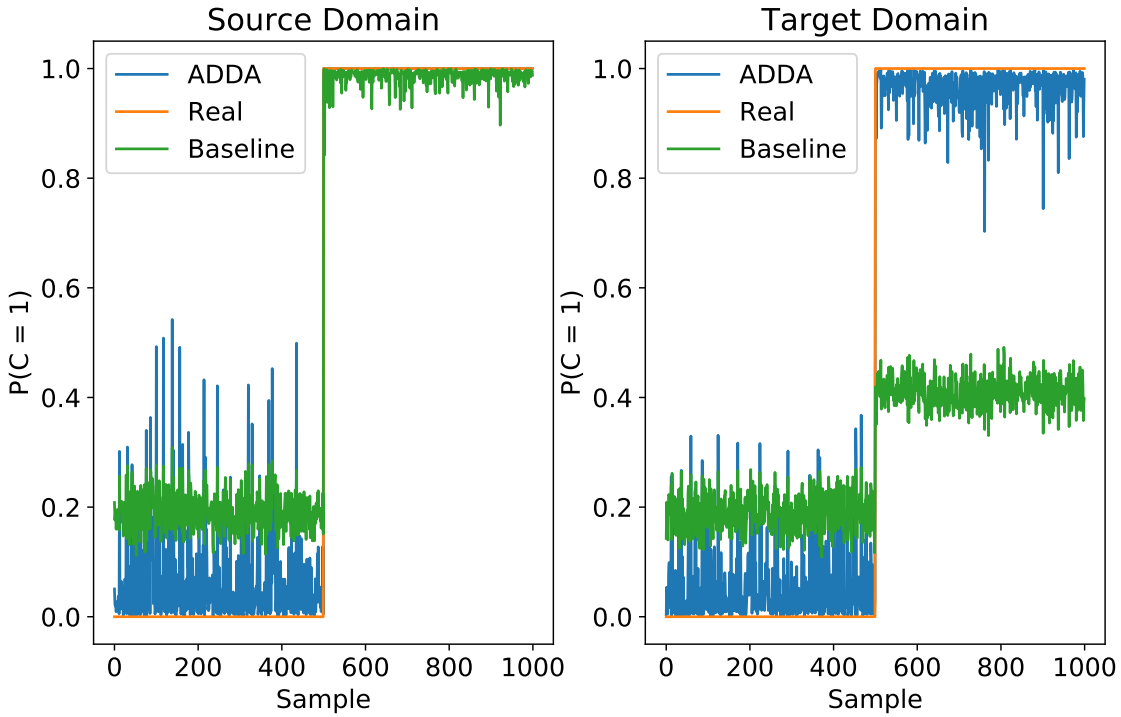

Figure 16: Results on the toy data domain adaptation dataset; the predicted probability $P(C = 1)$ on both domains for two models: one that is only trained on the source domain and one that uses the ADDA algorithm. Note how the Baseline algorithm predicts a constant class on the target domain ($P(C = 1) < 0.5$), even though half the samples are from the other class.

convolutional layer (64 filters of size 15 and stride 2) with max pooling of size 6, that connects to a dense layer. We employ batch normalization during training, but no dropout. The encoder outputs a 50-dimensional feature vector to be used by both the discriminator and the classifier, which are modelled as simple dense neural networks with two and one layers, respectively.

The model is pretrained for 10 epochs using a batch size of 256 and the Adam optimizer, after which domain adaptation is performed for 20 epochs. As the loss function we use the cross-entropy loss. The baseline model is simply the source encoder trained during the pretraining step, which achieves 100% test accuracy on the source domain.

Figure 16 shows the results for this task. The plot gives the predicted probabilities for both classes on both domains, for two different models. The baseline model is simply the result of the pretraining step in the ADDA algorithm – it is only trained on the source domain data. On the other hand, the domain adaptation step employs the target domain data in an unsupervised manner. The baseline model performs perfectly on the source domain, but struggles to generalize when the underlying phenomenon stops being related to only the scale of the data. ADDA, on the other hand, learns features that exist in both domains, allowing it to generalize to the

target domain.

Our experiments show that, for data such as that resulting from ultrasonic cleaning, domain adaptation methods can be applied successfully. This is important because such data is always generated in the physical world, which causes different experiments to have different properties, and introduces various types of measurement noise into the data. In this thesis we did not run DA experiments on real data, because we lacked a dataset that was suitable for demonstrating its effectiveness.

6 Conclusions

The problem of cleaning industrial fouling is significant in terms of the global economy and global warming [41] and solving it efficiently is possible using ultrasonic cleaning. To make the technology even more energy efficient, machine learning should be used to analyze the phenomenon and detect when cleaning is not required. The goal of this thesis was to show that in a limited laboratory setting, it is possible to create models that achive this and are robust to changes in the equipment.

To this end, we built the first system in the world capable of concurrently cleaning the fouled laboratory equipment while detecting whether or not fouling is present. The system uses a machine learning model which is fed signals in real time, and is trained using historical cleaning data. A completely new CNN architecture was presented, based on recent work on high-dimensional signals [43], capable of generalizing to new experiments despite the presence of domain shift.

Our methods relied on the fact that the amount of fouling can only reduce during cleaning. As such, we only assumed that for our training data, the first k samples are fouled, and the last k samples are clean (with experiments provided for multiple k). This lead to a semi-supervised algorithm with multiple variants for fine control over our assumptions, which was based on previous work in semi-supervised high-dimensional time-series data [5]. The machine learning contributions presented here were also published in a scientific forum [6], and we expect the monotonicity supervision idea to be of use for any task where a quantity changes monotonically through time.

Further, we showed that a clear domain shift phenomena is present between the different sets of experiments taken at different times, and presented methods of countering the poor generalization it usually causes. While this turned out not to be useful for the data attained from our laboratory experiments, we showed on a similar, synthetic dataset that DA methods may be used to counter domain shift in high-dimensional signal data.

While our methods worked well in the laboratory setting, in the future we would like to test them on real industrial data. To this end, our experimental setup was slightly too simple – it only featured one internal pipe which was fouled, as opposed to the tens of internal structures a real system might have. We speculate that in real equipment the need for DA will become more important, since there are far more changing variables in terms of the environment that hosts the equipment.

In addition, our equipment was made of acrylic instead of steel or other materials. Our model did not make any assumptions about the make of the equipment, so in theory it should generalize to steel, but this remains to be seen on actual data. Our CNN model also features a very low amount of parameters, which while very robust, might become a problem when more complex equipment is involved.

Another key point about the experimental setup is that only a single, small radius

in the pipe is being monitored with ultrasound, which means that if the single point becomes clean before the rest of the equipment does, our method will classify the equipment as clean in total. In future work, one possible solution would be to add more measurement transducers for monitoring.

To conclude, as a result of our work it is possible to detect fouling during ultrasonic cleaning with good accuracy. In particular, our method is no more than 10 seconds off on average from the ground truth point of cleaning, a minuscule time in terms of the total time industrial equipment in run for. While further work is clearly needed to address the shortcomings of this work, we expect it to have an impact in the field of industrial fouling detection.

References

1 E. Wallhäußer, M. Hussein, and T. Becker, "Detection methods of fouling in heat exchangers in the food industry," *Food Control*, vol. 27, no. 1, pp. 1–10, 2012.

2 "Altum technologies." `http://www.altumtechnologies.com`. Online; accessed 7 August 2018.

3 Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

4 D.-H. Lee, "Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks," in *Workshop on Challenges in Representation Learning, ICML*, vol. 3, p. 2, 2013.

5 K. Longi, T. Pulkkinen, and A. Klami, "Semi-supervised convolutional neural networks for identifying wi-fi interference sources," in *Proceedings of the Ninth Asian Conference on Machine Learning* (M.-L. Zhang and Y.-K. Noh, eds.), vol. 77 of *Proceedings of Machine Learning Research*, pp. 391–406, PMLR, 15–17 Nov 2017.

6 C. Rajani, A. Klami, A. Salmi, T. Rauhala, E. Hæggström, and P. Myllymäki, "Detecting industrial fouling by monotonicity during ultrasonic cleaning," in *IEEE Workshop on Machine Learning for Signal Processing, 2018*, pp. –, IEEE, 2018.

7 K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

8 K. P. Murphy, "Machine learning: a probabilistic perspective. 2012," *Cité en*, p. 117, 2014.

9 A. Karpathy, "Cs231n convolutional neural networks for visual recognition," *Neural networks*, vol. 1, 2016.

10 X. Zhu, "Semi-supervised learning," in *Encyclopedia of machine learning*, pp. 892–897, Springer, 2011.

11 A. Rasmus, M. Berglund, M. Honkala, H. Valpola, and T. Raiko, "Semi-supervised learning with ladder networks," in *Advances in Neural Information Processing Systems*, pp. 3546–3554, 2015.

12 A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

13 O. Boiman, E. Shechtman, and M. Irani, "In defense of nearest-neighbor based image classification," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pp. 1–8, IEEE, 2008.

14 I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

15 J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

16 D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

17 N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

18 L. B. Rall and G. F. Corliss, "An introduction to automatic differentiation," *Computational Differentiation: Techniques, Applications, and Tools*, vol. 89, 1996.

19 M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: a system for large-scale machine learning.," in *OSDI*, vol. 16, pp. 265–283, 2016.

20 A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

21 V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

22 K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

23 J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

24 N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

25 S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, pp. 448–456, 2015.

26 S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, "How does batch normalization help optimization?(no, it is not about internal covariate shift)," *arXiv preprint arXiv:1805.11604*, 2018.

27 G. E. Hinton and S. T. Roweis, "Stochastic neighbor embedding," in *Advances in neural information processing systems*, pp. 857–864, 2003.

28 L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

29 J. Li, M.-T. Luong, and D. Jurafsky, "A hierarchical neural autoencoder for paragraphs and documents," *arXiv preprint arXiv:1506.01057*, 2015.

30 X. Lu, Y. Tsao, S. Matsuda, and C. Hori, "Speech enhancement based on deep denoising autoencoder.," in *Interspeech*, pp. 436–440, 2013.

31 J. Xie, L. Xu, and E. Chen, "Image denoising and inpainting with deep neural networks," in *Advances in neural information processing systems*, pp. 341–349, 2012.

32 M. Sugiyama, M. Krauledat, and K.-R. MÃžller, "Covariate shift adaptation by importance weighted cross validation," *Journal of Machine Learning Research*, vol. 8, no. May, pp. 985–1005, 2007.

33 B. Sun, J. Feng, and K. Saenko, "Return of frustratingly easy domain adaptation.," in *Association for the Advancement of Artificial Intelligence*, vol. 6, p. 8, 2016.

34 M. Long, Y. Cao, J. Wang, and M. Jordan, "Learning transferable features with deep adaptation networks," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 97–105, PMLR, 07–09 Jul 2015.

35 E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell, "Adversarial discriminative domain adaptation," *arXiv preprint arXiv:1702.05464*, 2017.

36 Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, "Domain-adversarial training of neural networks," *Journal of Machine Learning Research*, vol. 17, no. 59, pp. 1–35, 2016.

37 S. Motiian, Q. Jones, S. M. Iranmanesh, and G. Doretto, "Few-shot adversarial domain adaptation," 2017.

38 J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255, IEEE, 2009.

39 I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

40 H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, and M. Marchand, "Domain-adversarial neural networks," *arXiv preprint arXiv:1412.4446*, 2014.

41 H. Müller-Steinhagen, M. Malayeri, and A. Watkinson, "Fouling of heat exchangers-new approaches to solve an old problem," *Heat transfer engineering*, vol. 26, no. 1, pp. 1–4, 2005.

42 E. Wallhäußer, W. B. Hussein, M. A. Hussein, J. Hinrichs, and T. M. Becker, "On the usage of acoustic properties combined with an artificial neural network–a new approach of determining presence of dairy fouling," *Journal of Food Engineering*, vol. 103, no. 4, pp. 449–456, 2011.

43 H. Phan, L. Hertel, M. Maass, and A. Mertins, "Robust audio event recognition with 1-max pooling convolutional neural networks," *Interspeech 2016*, pp. 3653–3657, 2016.

44 M. Ayer, H. D. Brunk, G. M. Ewing, W. T. Reid, and E. Silverman, "An empirical distribution function for sampling with incomplete information," *The annals of mathematical statistics*, pp. 641–647, 1955.

45 Y.-L. Yu and E. P. Xing, "Exact algorithms for isotonic regression and related," in *Journal of Physics: Conference Series*, vol. 699, p. 012016, IOP Publishing, 2016.

46 H. Zhao, S. Zhang, G. Wu, J. P. Costeira, J. M. F. Moura, and G. J. Gordon, "Multiple source domain adaptation with adversarial training of neural networks," 2017.